Sébastien Hoarau
Thierry Massart
Jean Olgiati
Isabelle Poirier

Septembre 2023

Conditions d'utilisation du contenu du cours

CC-BY-SA : Sébastien Hoarau - Thierry Massart - Jean Olgiati - Isabelle Poirier

Attribution - Partage dans les Mêmes Conditions

Les contenus peuvent être partagés et adaptés, y compris dans un but commercial, sous réserve de créditer l'oeuvre originale et de partager l'oeuvre modifiée dans les mêmes conditions.

Table des matières

Avant Propos **I**

1 Bienvenue dans l'environnement Python 3 **3**
1.1 Première mise en contact . . . 3
1.1.1 Sondage de début de cours . . . 3
1.1.2 Bienvenue dans le cours en ligne : Apprendre à coder avec Python . . . 3
1.1.3 Vos motivations et combien de temps allons-nous passer ensemble . . . 5
1.2 Modalités pratiques du cours . . . 8
1.2.1 Contenu de cette section . . . 8
1.2.2 Informations générales sur le cours . . . 8
1.2.3 Les modalités d'évaluation de notre MOOC . . . 9
1.2.4 Naviguer dans notre MOOC . . . 9
1.2.5 Aperçu de la suite de ce module . . . 11
1.3 Mode d'emploi pour installer un environnement de développement . . . 12
1.3.1 Que va-t-on installer ? . . . 12
1.3.2 Utilisation de Thonny . . . 16
1.3.3 Si je veux quand même installer PyCharm plutôt que Thonny sur mon ordinateur Windows . . . 20
1.3.4 Si je veux quand même installer PyCharm plutôt que Thonny sur mon ordinateur MacOs . . . 21
1.3.5 Si je veux quand même installer PyCharm plutôt que Thonny sur mon ordinateur Ubuntu ou Linux . . . 22
1.3.6 Si l'installation ne fonctionne pas . . . 23
1.3.7 Installation d'un dictionnaire français dans PyCharm . . . 24
1.3.8 Utilisation de PyCharm . . . 25
1.4 UpyLaB, Python Tutor et la documentation officielle . . . 27
1.4.1 Notre exerciseur UpyLaB . . . 27
1.4.2 Exercice UpyLaB 1.1 - Vert . . . 28
1.4.3 L'outil Python Tutor . . . 30
1.4.4 La documentation officielle sur python.org . . . 31
1.5 Au fait, c'est quoi un code ou un programme ? . . . 31
1.5.1 Quelques définitions . . . 31
1.5.2 Quiz de fin de module . . . 32
1.6 Références et bilan du module . . . 33
1.6.1 Références et bilan . . . 33

2 Python comme machine à calculer et à écrire **35**
2.1 Tout un programme au menu . . . 35
2.1.1 Présentation du menu de ce module . . . 35
2.2 Python comme machine à calculer . . . 36
2.2.1 Valeurs et expressions . . . 36
2.2.2 L'arithmétique . . . 37

2.2.3 Tester l'arithmétique . . . 38
2.3 Python comme machine de traitement de texte . . . 39
2.3.1 Les expressions chaînes de caractères . . . 39
2.3.2 À vous de tester les chaînes de caractères . . . 41
2.4 Les variables pour changer . . . 41
2.4.1 Les variables . . . 41
2.4.2 Code avec variables . . . 43
2.5 PyCharm en mode script, entrées et sorties . . . 44
2.5.1 Manipuler des scripts avec PyCharm . . . 44
2.5.2 Python Tutor et les diagrammes d'état . . . 45
2.5.3 Commentons notre programme . . . 46
2.5.4 Réalisation des exercices UpyLaB du module . . . 49
2.5.5 Exercice UpyLaB 2.1 : Vert . . . 50
2.5.6 Exercice UpyLaB 2.2 : Vert . . . 51
2.5.7 Exercice UpyLaB 2.3 : Vert . . . 52
2.6 Quelques fonctions prédéfinies, les modules math et turtle . . . 53
2.6.1 Exemples d'utilisations de fonctions prédéfinies, et des modules math et turtle . . . 53
2.6.2 Quelques fonctions prédéfinies et fonctions turtle très utilisées . . . 56
2.6.3 Pavé hexagonale . . . 59
2.6.4 Exercice UpyLaB 2.4 - Non noté : Bleu . . . 61
2.7 Pour terminer ce module . . . 63
2.7.1 Stockage des valeurs et caractères d'échappement . . . 63
2.7.2 Les opérateurs d'assignation et de mise à jour . . . 65
2.7.3 On passe à la pratique autonome ! . . . 66
2.7.4 Exercice UpyLaB 2.5 : Vert . . . 66
2.7.5 Exercice UpyLaB 2.6 : Vert . . . 67
2.7.6 Exercice UpyLaB 2.7 : Bleu . . . 68
2.8 Quiz de fin et bilan du module . . . 69
2.8.1 Quiz de fin de module . . . 69

3 Les instructions : tous vos désirs sont des ordres 71
3.1 Presque toutes les instructions au menu . . . 71
3.1.1 Présentation du menu de ce module . . . 71
3.2 L'instruction conditionnelle if : fais pas « si », fais pas ça ! . . . 71
3.2.1 L'instruction if . . . 71
3.2.2 Les opérateurs relationnels . . . 73
3.2.3 Exemples de code avec des if . . . 74
3.2.4 Opérateurs logiques . . . 76
3.2.5 Syntaxe et sémantique de l'instruction if . . . 78
3.2.6 Quiz sur l'instruction if . . . 81
3.3 Pratique de l'instruction if . . . 81
3.3.1 Mettons ensemble en pratique ce que nous venons de voir . . . 81
3.3.2 Jeu de devinette : proposition de solution . . . 81
3.3.3 Le if : mise en pratique autonome : exercice UpyLaB 3.1 . . . 82
3.3.4 Exercice UpyLaB 3.2 (Vert) . . . 84
3.3.5 Exercice UpyLaB 3.3 (Vert) . . . 86
3.3.6 Exercice UpyLaB 3.4 (Vert) . . . 87
3.3.7 Exercice UpyLaB 3.5 (Vert) . . . 88
3.3.8 Exercice UpyLaB 3.6 (Vert) . . . 89
3.3.9 Exercice UpyLaB 3.7 (Vert) . . . 90
3.3.10 Exercice UpyLaB 3.8 (Rouge) . . . 92
3.4 Les instructions répétitives while et for . . . 95
3.4.1 L'instruction while . . . 95
3.4.2 Syntaxe du while et calcul du plus grand commun diviseur . . . 96
3.4.3 Conjecture de Syracuse . . . 97
3.4.4 Deux canevas classiques rencontrés avec une boucle while . . . 99

 CC BY-SA 3.0 - 2023 - Sébastien Hoarau - Thierry Massart - Jean Olgiati - Isabelle Poirier

3.4.5 Des programmes qui bouclent indéfiniment . . . 100
3.4.6 L'instruction for . . . 100
3.4.7 Syntaxe du for, carrés, étoiles et autres polygones réguliers . . . 102
3.4.8 Quiz sur while et for . . . 104
3.5 Code avec while et for dans la pratique . . . 104
3.5.1 La suite de Fibonacci . . . 104
3.5.2 Un pavé hexagonal avec des for . . . 107
3.5.3 Mise en pratique autonome : exercices UpyLaB 3.9 et suivants . . . 108
3.5.4 Exercice UpyLaB 3.10 (Vert) . . . 110
3.5.5 Note sur la fonction print . . . 111
3.5.6 Exercice UpyLaB 3.11 (Vert) . . . 112
3.5.7 Exercice UpyLaB 3.12 (Bleu) . . . 113
3.5.8 Exercice UpyLaB 3.13 (Bleu) . . . 114
3.5.9 Exercice UpyLaB 3.14 (Rouge) . . . 115
3.5.10 Exercice UpyLaB 3.15 (Rouge) . . . 117
3.5.11 range(debut, fin, pas) . . . 120
3.5.12 Exercice UpyLaB 3.16 (Rouge) . . . 122
3.5.13 Exercice UpyLaB 3.17 (Rouge) . . . 123
3.5.14 Exercice UpyLaB 3.18 (Noir) . . . 124
3.6 L'instruction pass et quiz de fin de module . . . 126
3.6.1 L'instruction pass . . . 126
3.6.2 Quiz de fin de module . . . 126
3.7 Bilan du module . . . 127
3.7.1 Qu'avons-nous vu dans ce module ? . . . 127

4 Les fonctions : créons les outils que nous voulons 129
4.1 Au menu : comment fonctionne une fonction ? . . . 129
4.1.1 Présentation du menu de ce module . . . 129
4.2 Les fonctions prédéfinies et définies . . . 129
4.2.1 Utilisation et définition de fonctions . . . 129
4.2.2 Définition de nouvelles fonctions . . . 131
4.2.3 Exécution de fonctions . . . 132
4.2.4 Précisions sur les fonctions . . . 133
4.2.5 Mini quiz . . . 135
4.3 Faisons fonctionner les fonctions . . . 135
4.3.1 Autres exemples de fonctions . . . 135
4.3.2 Quelques aspects plus techniques . . . 138
4.3.3 Première mise en pratique des fonctions . . . 142
4.3.4 Mise en pratique des fonctions : exercices UpyLaB 4.1 et suivants . . . 144
4.3.5 Exercice UpyLaB 4.2 (Vert) . . . 145
4.3.6 Exercice UpyLaB 4.3 (Bleu) . . . 149
4.4 Quelques règles de bonnes pratiques . . . 151
4.4.1 Quiz "Testez vos connaissances !" . . . 151
4.4.2 Code ou programme « propre » . . . 151
4.4.3 Quelques éléments supplémentaires . . . 155
4.4.4 Petit manuel des règles de bonnes pratiques . . . 157
4.4.5 La règle du return unique et les instructions dont on ne veut pas prononcer le nom . . . 157
4.5 Pratique des fonctions . . . 159
4.5.1 Mettons ensemble en pratique ce que nous venons de voir . . . 159
4.5.2 Autre exercice accompagné . . . 160
4.5.3 Le yin et le yang revisité et paramètres par défaut . . . 161
4.5.4 Un pavé hexagonal avec fonctions . . . 164
4.5.5 Mise en pratique : exercice UpyLaB 4.4 et sq . . . 165
4.5.6 Exercice UpyLaB 4.5 (Vert) . . . 167
4.5.7 Exercice UpyLaB 4.6 (Vert) . . . 168
4.5.8 Exercice UpyLaB 4.7 (Bleu) . . . 169

4.5.9 Exercice UpyLaB 4.8 (Bleu) . . . 171
4.5.10 Exercice UpyLaB 4.9 (Bleu) . . . 173
4.5.11 Exercice UpyLaB 4.10 (Rouge) . . . 175
4.6 Bilan du module . . . 178
4.6.1 Qu'avons-nous vu dans ce module ? . . . 178

5 Les séquences de données pour jouer ou travailler 181
5.1 Manipulations de base des séquences . . . 181
5.1.1 Présentation du menu de ce module et des chaînes de caractères . . . 181
5.1.2 Les tuples et les listes . . . 182
5.1.3 Tests sur les séquences et instruction for . . . 183
5.1.4 Encodage des caractères . . . 185
5.1.5 Illustrons ce que nous avons vu jusqu'à présent sur les séquences . . . 186
5.1.6 Mise en pratique des séquences : exercices UpyLaB 5.1 et suivants . . . 188
5.1.7 Exercice UpyLaB 5.2 (Vert) . . . 189
5.1.8 Exercice UpyLaB 5.3 (Vert) . . . 190
5.1.9 Exercice UpyLaB 5.4 (Vert) . . . 191
5.1.10 Rassembler et séparer . . . 192
5.1.11 Exercice UpyLaB 5.5 (Bleu) . . . 194
5.2 Slicing et opérations de manipulation des séquences . . . 195
5.2.1 Tranches, manipulations et séquences imbriquées . . . 195
5.2.2 Suite sur les séquences de données . . . 196
5.2.3 Seconde suite sur les séquences de données . . . 198
5.2.4 Manipulations simples de séquences . . . 200
5.2.5 À vous de jouer avec les exercices UpyLaB 5.6 et suivants ! . . . 205
5.2.6 Exercice UpyLaB 5.7 (Vert) . . . 206
5.2.7 Exercice UpyLaB 5.8 (Bleu) . . . 208
5.3 Testons vos connaissances sur les séquences . . . 209
5.3.1 Quiz sur les séquences . . . 209
5.4 Méthodes de manipulation des séquences . . . 209
5.4.1 Présentation d'autres méthodes de manipulation des séquences . . . 209
5.4.2 Manipulation de chaînes de caractères . . . 211
5.4.3 Manipulation de listes . . . 213
5.4.4 Exemples simples qui utilisent des méthodes sur des séquences . . . 215
5.4.5 Mise en pratique : exercices Upylab 5.9 et suivants avec ou sans méthode ! . . . 216
5.4.6 Exercice UpyLaB 5.10 (Vert) . . . 218
5.4.7 Exercice UpyLaB 5.11 (Bleu) . . . 219
5.4.8 Exercice UpyLaB 5.12 (Bleu) . . . 220
5.4.9 Exercice UpyLaB 5.13 (Rouge) . . . 221
5.4.10 Exercice UpyLaB 5.14 (Vert) . . . 222
5.4.11 Exercice UpyLaB 5.15 (Rouge) . . . 224
5.5 Testons nos connaissances . . . 225
5.5.1 Quiz sur les séquences . . . 225
5.6 Compréhensions et copies de listes . . . 225
5.6.1 Concepts de compréhension et copie profonde de liste . . . 225
5.6.2 Exemples avec compréhensions de listes . . . 227
5.6.3 Pratiquons encore : exercices UpyLaB 5.16 et suivant . . . 227
5.6.4 Exercice UpyLaB 5.16 (Vert) . . . 227
5.6.5 Exercice UpyLaB 5.17 (Bleu) . . . 229
5.6.6 Exercice UpyLaB 5.18 (Rouge) . . . 229
5.7 Testons nos connaissances . . . 231
5.7.1 Un petit quiz pour consolider tout cela . . . 231
5.8 Lisons et sauvons les textes . . . 231
5.8.1 Manipulation de fichiers textes . . . 231
5.8.2 Manipulation de fichiers textes (suite) . . . 232
5.8.3 Les fichiers textes (suite) . . . 234

 CC BY-SA 3.0 - 2023 - Sébastien Hoarau - Thierry Massart - Jean Olgiati - Isabelle Poirier

5.8.4 Des exemples de manipulation de fichiers . . . 235
5.8.5 Les traitements de fichiers avec UpyLaB 5.19 et suivants . . . 237
5.8.6 Exercice UpyLaB 5.20 (Vert) . . . 238
5.8.7 Exercice UpyLaB 5.21 (Bleu) . . . 239
5.8.8 Exercice upylab 5.22 (Rouge) . . . 240
5.9 Manipulons les matrices . . . 241
5.9.1 Remarque importante sur l'initialisation des tableaux . . . 241
5.9.2 À présent exerçons-nous sur les matrices . . . 242
5.9.3 Passons à la mise en pratique sur les matrices avec les exercices UpyLaB 5.26 et suivants . . . 244
5.9.4 Exercice UpyLaB 5.24 (Vert) . . . 246
5.9.5 Exercice UpyLaB 5.25 (Bleu) . . . 246
5.9.6 Exercice UpyLaB 5.26 (Bleu) . . . 247
5.9.7 Exercice UpyLaB 5.27 (Rouge) . . . 249
5.10 Bilan du module . . . 250
5.10.1 Qu'avons-nous vu dans ce module ? . . . 250

6 Ensembles et dictionnaires : armes de construction massive 253
6.1 Manipulations de base des séquences ensembles et dictionnaires . . . 253
6.1.1 Introduction aux ensembles et aux dictionnaires . . . 253
6.1.2 Les dictionnaires . . . 255
6.1.3 Petits exemples manipulant ensembles et dictionnaires . . . 257
6.1.4 Mise en pratique des ensembles et dictionnaires avec les exercices UpyLaB 6.1 et suivants . . . 259
6.1.5 EXERCICE UPYLAB 6.2 (Vert) . . . 260
6.1.6 Exercice UpyLaB 6.3 (Bleu) . . . 261
6.1.7 Exercice UpyLaB 6.4 (Bleu) . . . 262
6.1.8 Exercice UpyLaB 6.5 (Vert) . . . 263
6.1.9 Exercice UpyLaB 6.6 (Rouge) . . . 264
6.1.10 Exercice UpyLaB 6.7 (Vert) . . . 265
6.2 Petit quiz sur les ensembles et les dictionnaires . . . 266
6.2.1 Mini quiz sur les ensembles et les dictionnaires . . . 266
6.3 Méthodes de manipulation des dictionnaires . . . 266
6.3.1 Présentons les méthodes des dictionnaires . . . 266
6.3.2 Petits exemples manipulant des méthodes de dictionnaires . . . 269
6.3.3 Exercice UpyLaB 6.8 (Vert) . . . 270
6.3.4 Exercice UpyLaB 6.9 (Bleu) . . . 271
6.3.5 Exercice UpyLaB 6.10 (Rouge) . . . 272
6.4 Quiz de synthèse de la matière . . . 273
6.4.1 Quiz de synthèse sur les structures de données . . . 273
6.5 Exercices de synthèse . . . 273
6.5.1 Des exercices UpyLaB qui mobilisent plusieurs matières . . . 273
6.5.2 Exercice UpyLaB 6.11 (Vert) . . . 274
6.5.3 Exercice UpyLaB 6.12 (Vert) . . . 275
6.5.4 Exercice UpyLaB 6.13 (Vert) . . . 275
6.5.5 Exercice UpyLaB 6.14 (Vert) . . . 277
6.5.6 Exercice UpyLaB 6.15 (Rouge) . . . 278
6.5.7 Exercice UpyLaB 6.16 (Rouge) . . . 280
6.5.8 Exercice UpyLaB 6.17 (y compris 6.17a, 6.17b et 6.17c) (Bleu) . . . 282
6.5.9 Exercice UpyLaB 6.18 (Noir) . . . 286
6.5.10 EXERCICE UPYLAB 6.19 (Rouge) . . . 288
6.5.11 EXERCICE UPYLAB 6.20 (Noir) . . . 291
6.6 Données persistantes . . . 293
6.6.1 Présentons le format csv . . . 293
6.6.2 Présentons le module json . . . 294
6.7 Bilan du module . . . 296
6.7.1 Qu'avons-nous vu ? . . . 296
6.7.2 Et maintenant ? . . . 297

6.8 Bilan global du cours . . . 298
6.8.1 Qu'avons nous appris dans ce cours ? . . . 298
6.8.2 Quelles suites possibles à ce cours ? . . . 298

7 Votre projet **301**
7.1 Quête dans le château au sommet du Python des Neiges . . . 301
7.1.1 Information préliminaire . . . 301
7.1.2 Présentation du projet . . . 302
7.1.3 En pratique . . . 305
7.2 Détails des phases du projet . . . 308
7.2.1 Niveau 1 : construction et affichage du plan du château . . . 308
7.2.2 Niveau 2 : gestion des déplacements . . . 310
7.2.3 Niveau 3 : collecte d'objets dans le labyrinthe . . . 312
7.2.4 Niveau 4 : Le jeu escape game complet avec questions-réponses . . . 313
7.2.5 Remise et évaluation du projet . . . 313
7.3 Phase d'évaluation par les pairs . . . 316
7.3.1 Phase d'évaluation par les pairs . . . 316
7.4 Phase d'auto-évaluation du projet . . . 317
7.4.1 Phase d'auto-évaluation . . . 317

8 Projet donné lors des trois premières sessions du MOOC Apprendre à coder avec Python **319**
8.1 Le projet Vasarely . . . 319
8.1.1 Énoncé du projet . . . 319
8.1.2 En pratique . . . 321
8.1.3 Remise et évaluation du projet . . . 326
8.1.4 Galerie des oeuvres produites par le projet . . . 329

9 Annexes **331**

Index **331**

CC BY-SA 3.0 - 2023 - Sébastien Hoarau - Thierry Massart - Jean Olgiati - Isabelle Poirier

Avant Propos

Bienvenue dans la session 6 (2022-2023) de notre MOOC intitulé Apprendre à coder avec Python diffusé sur la plateforme FUN.

Cette version du cours est le fruit d'un travail important que nous avons réalisé depuis la première session diffusée en 2019.

Depuis la session 5 (2021-2022), nous avons apporté les améliorations ou ajouts suivants :
- Suppression de la notion de parcours et identification de couleur pour chaque exercice (vert, bleu, rouge, noir) en fonction de sa difficulté ;
- Solution type de certains exercices après validation de ceux-ci par l'apprenant ;
- Amélioration de certaines parties du cours.

Depuis la session 4 (2020-2021), nous avons apporté les améliorations ou ajouts suivants :

Contenu

Ce cours intègre désormais :
- un manuel comme support écrit pour accompagner le MOOC ;
- des sous-titres aux vidéos (dans la version en ligne), activables lors de la visualisation (attention : ne fonctionne pas avec le navigateur Safari) ;
- une Foire Aux Questions (FAQ) pour vous permettre de trouver une réponse aux questions fréquemment posées ;
- des résumés des vidéos du cours ;
- des énoncés de notre exerciseur UpyLaB, mieux structurés avec des exemples de ce qui est demandé ;
- un choix des exercices UpyLaB encore mieux adapté au cours ;

Par ailleurs, cette session vous propose un projet (« Quête au Château du sommet du Piton Rocheux ») où il vous sera demandé de construire et de jouer à votre jeu d'évasion (escape game) sur ordinateur.

Horaire de diffusion

Nous avons également aménagé l'horaire de diffusion pour vous permettre de suivre ce cours soit sur un semestre (septembre-décembre ou janvier-mai), soit sur toute l'année scolaire.

Nous vous invitons à vous inscrire dès à présent à cette nouvelle session pour bénéficier de l'ensemble des ressources.

Bon travail et bon amusement

Isabelle - Sébastien - Thierry

Septembre 2022

CHAPITRE 1

Bienvenue dans l'environnement Python 3

1.1 Première mise en contact

1.1.1 Sondage de début de cours

VOTRE AVIS NOUS INTÉRESSE

Note : Voir le sondage officiel FUN de la section 1.1.1

1.1.2 Bienvenue dans le cours en ligne : Apprendre à coder avec Python

QUI SOMMES-NOUS ET COMMENT ALLONS-NOUS VOUS APPRENDRE À CODER ?

Note : Voir la vidéo de la section 1.1.2 : Bienvenue dans le MOOC Apprendre à Coder avec Python

CONTENU DE LA VIDÉO

La vidéo précédente présente brievement nos motivations pour vous apprendre à coder avec le langage de programmation Python et l'approche proposée dans ce cours.

Tout ceci sera présenté plus en détails dans la suite de ce module de cours.

Cette version du cours est le fruit d'un travail important que nous avons réalisé depuis la première session diffusée en 2019.

MODIFICATIONS DEPUIS LA SESSION 5

Depuis la session 5 (2021-2022), nous avons apporté les améliorations ou ajouts suivants :

Suppression de la notion de parcours et identification de couleur pour chaque exercice (vert, bleu, rouge, noir) en fonction de sa difficulté ;

Solution type de certains exercices après validation de ceux-ci par l'apprenant ;

Amélioration de certaines parties du cours.

MODIFICATIONS DEPUIS LA SESSION 4

Contenu

Ce cours intègre désormais :

- un manuel comme support écrit pour accompagner le MOOC ;
- des sous-titres aux vidéos, activables lors de la visualisation (attention : ne fonctionne pas avec le navigateur Safari) ;
- une Foire Aux Questions (FAQ) pour vous permettre de trouver une réponse aux questions fréquemment posées ;
- des résumés des vidéos du cours ;
- des énoncés de notre exerciseur UpyLaB, mieux structurés avec des exemples de ce qui est demandé ;
- un choix des exercices UpyLaB encore mieux adapté au cours ;

Par ailleurs, cette session vous propose un projet (« Quête au Château du sommet du Piton Rocheux ») où il vous sera demandé de construire et de jouer à votre jeu d'évasion (escape game) sur ordinateur.

Horaire de diffusion

Nous avons également aménagé l'horaire de diffusion pour vous permettre de suivre ce cours soit sur un semestre (septembre-décembre ou janvier-mai), soit sur toute l'année scolaire.

Modules disponibles dès l'ouverture

- Le module 1 - *Bienvenue dans l'environnement Python 3* présente tout d'abord les éléments importants de ce MOOC, à la fois ce qui va être enseigné et comment vous allez l'apprendre. La seconde partie du module 1 vous demande d'installer l'environnement de développement Python 3 sur votre ordinateur en expliquant les procédures à suivre : c'est une étape essentielle à votre apprentissage.
- Le Module 2 - *Python comme machine à calculer et à écrire* va vous apprendre comment écrire votre premier programme Python 3 complet.
- Le Module 3 - *Les instructions : tous vos désirs sont des ordres* introduit les instructions de contrôle de flux Python, qui vont nous permettre d'écrire des codes bien plus sophistiqués.
- Le Module 4 - *Les fonctions : créez les outils que vous voulez* présente dans un premier temps des fonctions prédéfinies utiles. Ensuite nous allons regarder comment définir et utiliser nos propres fonctions Python. Nous discutons aussi sur la qualité d'un code, ce que nous appelons un code « propre ».

Bon travail et surtout bon amusement dans l'apprentissage de cette nouvelle matière !

CRÉDIT : NOTRE OUTIL UPYLAB

Dans la suite du module nous présentons notre outil UpyLaB que vous utiliserez de façon intégrée dans ce MOOC pour réaliser de nombreux exercices de codage.

UpyLaB est le fruit de nombreuses années de développements d'informaticiens et étudiants de l'Université Libre de Bruxelles. Une bonne partie des développements et la coordination ont été réalisées de main de maître par notre informaticien, **Arthur Lesuisse**, que nous remercions vivement pour cela !

CRÉDIT : ÉQUIPES PEDAGOGIQUES ET MULTIMEDIA

Équipe CAP-ULB (anciennement « ULB-PODCAST » de l'ULB

Ce MOOC n'aurait pas vu le jour sans les membres de l'équipe ULB-Podcast qui nous ont accompagnés durant toutes les étapes, de sa conception à sa diffusion :

— Ariane Bachelart : accompagnement pédagogique et technique FUN
— Guillaume Gabriel : production multimédia
— Jérôme Di Egidio : production audiovisuelle
— Nathalie François : conseils pédagogiques
— Sophie Poukens : accompagnement pédagogique

Équipe de la Direction des Usages du Numérique de l'UR

Nous voulons également remercier les membres de l'équipe de la Direction des Usages du Numérique de l'Université de la Réunion (DUN) :

— Emmanuel Pons : production audiovisuelle
— Jean-François Février : production audiovisuelle

Co-concepteur du nouveau projet « Quête au Château du sommet du Piton Rocheux »

Merci également à Jean Olgiati qui a co-conçu le nouveau projet « Quête au Château au sommet du Piton Rocheux ».

Bêta-testeurs et participants au teaser et à la première session

Merci aussi aux « Bêta-testeurs » de la première heure qui nous ont conseillés dans la réalisation de ce MOOC ainsi qu'aux personnes qui ont participé aux séances de confection du « teaser » : Ali Manzer, Anne-Françoise Biet, Arnaud Decostre, Ashley Yong Ching, Fabrice Nativel, Françoise Bols, Ludovic De Wachter, Marco Bianchin, Nicolas Pettiaux, Pierre Evrard, Priscillia Hardas, Ronico Billy, Vincenzo Sargenti, Yishen Yu.

Merci aussi aux participants des premières sessions de ce MOOC qui, par leurs commentaires, nous ont aidés à améliorer ce cours. Merci à tous !

1.1.3 Vos motivations et combien de temps allons-nous passer ensemble

NOS MOTIVATIONS

Nos premières motivations pour donner ce cours sont de vous montrer que programmer, c'est créer, et de vous transmettre le bonheur de cette activité.

CLÉ POUR RÉUSSIR CE COURS

Vous allez apprendre à programmer en utilisant le langage Python 3. Contrairement à d'autres matières comme l'apprentissage d'une langue étrangère ou des mathématiques, il s'agit probablement de quelque chose de réellement neuf pour vous.

L'apprentissage peut être comparé à l'apprentissage de la marche chez un enfant :

— L'enfant qui apprend à marcher va d'abord jouer dans son parc ou son bac à sable ; cela lui permet d'apprendre des gestes et de réaliser des choses utiles (ou non) pour cet apprentissage spécifique.
— Ensuite, quelqu'un va peut-être lui expliquer comment faire pour se lever, avancer, ...
— Plus tard, il va être aidé pour commencer à faire ses premiers pas.
— Et enfin, il va marcher et ensuite se lever seul et s'entrainer dur pour s'améliorer.

Nous vous invitons à faire le même type de parcours ici pour votre apprentissage du codage avec Python.

Pour réussir, il faut rigueur et persévérance et vous devez répondre oui aux questions suivantes :

1) Pourrez-vous consacrer suffisamment de temps, en respectant les échéances, pour cet apprentissage incluant visionnage des vidéos, lecture des explications, réponses aux quiz après mûres réflexions et codage des exercices demandés (entre 100 et 200 heures de travail, parfois plus chez certains) ?
2) Êtes-vous prêt à apprendre une matière fort différente de ce que vous avez appris jusqu'à présent ?
3) Pourrez-vous, même si notre équipe et la communauté des apprenants seront présentes sur le forum de discussion pour répondre à vos questions, avoir assez d'autonomie pour mener à bien cet apprentissage ?
4) Êtes-vous conscient que la persévérance et la résilience sont deux clés fondamentales pour apprendre ?

MAIS QUELLES SONT VOS MOTIVATIONS POUR SUIVRE CE COURS ?

Et vous, pouvez-vous résumer vos motivations à suivre ce cours ? Écrivez dans les rectangles ci-dessous au maximum quatre mots ou phrases courtes (un par rectangle) qui représentent le mieux vos motivations et cliquez sur "enregistrer". Vous pourrez alors voir ce que les autres apprenants ont répondu.

Note : Le sondage est diponible en section 1.1.3 du cours en ligne

---

QUEL TEMPS POUVEZ-VOUS CONSACRER À CE COURS EN LIGNE ?

ET QUEL PARCOURS ALLEZ-VOUS SUIVRE ?

Ce cours en ligne, appelé aussi MOOC, est construit sur une base d'une dizaine de semaines de travail étalées sur 15 semaines. MOOC est l'acronyme de « Massive Open Online Course », prononcé « mouc », c'est-à-dire un cours ouvert, gratuit et massif en ligne.

Apprendre à programmer peut être fortement chronophage : le cours complet demande donc un rythme assez soutenu. Si vous ne pouvez ou ne voulez pas consacrer autant de temps chaque semaine à cet apprentissage, nous vous proposons un aménagement que vous pourrez éventuellement choisir de suivre n'importe quand tout au long du cours.

A priori le cours demande environ 12 heures de travail chaque semaine sur 10 à 15 semaines. C'est une évaluation moyenne. En réalité le temps que vous mettrez dépendra essentiellement de votre facilité à assimiler la matière et à réaliser les exercices, principalement de notre exerciseur UpyLaB.

De toute manière, si vous ne pouvez réaliser le travail complet, nous vous proposons de sélectionner les exercices de codage (avec notre exerciseur UpyLaB).

Le principe est simple : nous vous conseillons de voir tous les contenus théoriques, les exercices solutionnés et de réaliser tous les quiz. Par contre, libre à vous de sélectionner parmi les exercices de codage de notre exerciseur UpyLaB, sachant que chaque exercice a un attribut de couleur qui correspond à son niveau de difficulté estimé.

Un exercice :
- vert est un exercice de base ;
- bleu est un exercice plus difficile ;
- rouge est un exercice jugé difficile ;
- noir est un des exercices les plus diffiles du cours.

Libre à vous de moduler les exercices UpyLaB que vous réalisez, sachant que :
- Si vous ne faites que les exercices verts, le travail moyen estimé est d'environ 6 heures par semaine sur 15 semaines en supposant également que vous ne réalisez pas le projet final inclus dans le cours.
- Si vous faites tous les exercices verts et bleus, le travail moyen estimé sera d'environ 9 heures par semaine sur 15 semaines y compris ou non, un projet final ainsi qu'un travail d" « évaluation par les pairs » du projet de trois autres apprenants, et une auto-évaluation de votre projet.

Note : Veillez à ne pas répondre aux quiz trop rapidement, puisque nombre d'entre eux ne permettent qu'un seul essai avant de noter votre réponse. Si votre réponse est fausse vous aurez alors définitivement perdu les points associés !

Enfin, si vous ne pouvez consacrer six heures par semaine sur 15 semaines (3 heures par semaines sur l'année scolaire), ou simplement si voulez suivre le cours et faire les exercices à un autre rythme, vous pouvez choisir de réaliser les exercices de votre choix à votre allure, sachant que vous pouvez obtenir une attestation FUN si vous avez obtenu 50% des points à la fin de ce cours, que le matériel et les exercices restent disponibles après la fin de la session, mais que les forums ne seront animés que pendant la période d'ouverture de la session et qu'il ne vous sera pas possible de soumettre le projet sur la plateforme ni d'obtenir l'attestation finale de succès après la clôture du MOOC.

CONSEILS POUR RÉUSSIR DANS VOTRE APPRENTISSAGE

— Soyez régulier dans votre apprentissage, par exemple en y consacrant une ou deux heures chaque jour.
— Si vous êtes bloqué dans la réalisation d'un exercice, n'hésitez pas à bien relire l'énoncé précis, à revoir encore et encore la matière déjà vue pour comprendre ce qui vous a échappé.
— La FAQ (Foire Aux Questions), organisée sous forme de questions d'apprenants et de réponses associées, est également une source d'information précieuse.
— Comme disait Nicolas Boileau en 1674 dans *l'Art Poétique* :

```
Vingt fois sur le métier remettez votre ouvrage,
Polissez-le sans cesse, et le repolissez,
Ajoutez quelquefois, et souvent effacez.
```

— Ne vous découragez pas et si rien ne va, adressez-vous aux forums pour chercher de l'aide.

COMBIEN D'HEURES PAR SEMAINE POURREZ-VOUS CONSACRER ?

— de 1 à 4 heures sur un semestre ou 1 heure pendant toute l'année scolaire
— 5 à 7 heures sur un semestre ou 2 à 3 heures pendant toute l'année scolaire
— 8 à 10 heures sur un semestre ou 4 à 5 heures pendant toute l'année scolaire
— 11 heures ou plus sur un semestre ou 6 heures ou plus pendant toute l'année scolaire

Choisissez l'option qui vous convient.
— Vous avez répondu « de 5 à 7 h / semestre ou 2 à 3 h / an » : faites en priorité les exercices UpyLaB verts
— Vous avez répondu de 8 à 10 h / semestre ou 4 à 5 h / an : faites en priorité les exercices UpyLaB verts et bleus
— Vous avez répondu 11 h ou + / semestre ou 6 h ou + / an : faites les exercices que vous souhaitez, dans l'ordre que vous désirez et essayez de faire le projet du cours qui est un véritable plus pour l'apprentissage
— Vous avez répondu de 1 à 4 h / semestre ou 1 h / an : faites les exercices que vous désirez sachant les exercices UpyLaB verts sont les plus simples et que pour obtenir une attestation de réussite FUN, vous devez avoir 50% des points à la clôture officielle du cours.

ET POURQUOI NE PAS NOUS EN DIRE UN PEU PLUS SUR LE FORUM ?

Nous allons passer du temps tous ensemble, pourquoi ne pas venir vous présenter dans le forum afin de faire connaissance ?

Venez nous raconter ce que vous faites dans la vie et pourquoi il est important pour vous de suivre ce cours.

Vous avez déjà utilisé le langage Python ou vous vous êtes déjà essayé à un langage informatique ? Partagez votre expérience avec la communauté.

ATTENTION !

Avant d'utiliser les forums, nous vous encourageons à prendre connaissance des règles et bonnes pratiques des forums via le lien https://www.fun-mooc.fr/asset-v1:ulb+44013+session05+type@asset+block@BonnesPratiquesForum.pdf et à les suivre scrupuleusement.

En particulier, ne mettez jamais d'informations personnelles comme votre nom, email ou n° de téléphone. Votre pseudo sera suffisant pour vous identifier.

Note : Voir le forum *Faisons connaissance* dans la section 1.1.3 du cours en ligne

---

1.2 Modalités pratiques du cours

1.2.1 Contenu de cette section

CONTENU DE CETTE SECTION

Dans cette section, nous donnons des informations générales sur notre MOOC (objectifs, calendrier, modalités d'évaluation), sur la manière d'utiliser la plateforme d'apprentissage FUN ainsi que les modalités de communication entre apprenants et l'équipe.

La section est terminée par une vidéo qui fait une rapide présentation de la suite de ce présent module.

1.2.2 Informations générales sur le cours

FICHE DU COURS ET CALENDRIER

Cette section reprend ou donne accès aux informations générales du cours y compris l'horaire de diffusion des différents modules et les échéances principales.

Fiche du cours

Elle donne la plupart des informations générales sur le cours, et est accessible via le lien que vous pouvez trouver ici (https://www.fun-mooc.fr/courses/course-v1:ulb+44013+session06/about)

Attestation de suivi avec succès

Tout apprenant atteignant un taux de réussite de 50% des points recevra une attestation de suivi avec succès produite et envoyée par la plateforme FUN.

Calendrier et échéances principales

Le calendrier et échéancier du MOOC est donné dans l'onglet Calendrier.

Vous pouvez également visualiser le calendrier en cliquant sur le lien ici (https://www.fun-mooc.fr/courses/course-v1:ulb+44013+session06/pdfbook/1/).

En particulier, vous devez impérativement respecter les trois échéances suivantes pour valider votre apprentissage :

— réalisation des quiz et exercices UpyLaB des modules 1 à 6 avant l'échéance du 22/12/2022 pour la première génération des attestations de suivi, ou pour l'échéance finale du 18/05/2023.

Si vous décidez de faire le projet :

Deux vagues d'évaluation vous sont proposées.

— remise du projet proposé avant le 17/11/2022 pour la première vague ou avant le 20/04/2023 pour la seconde vague ;
— évaluation de projets de pairs avant le 1/12/2022 pour la première vague ou avant le 4/05/2023 pour la seconde vague ;
— auto-évaluation de votre projet amendé avant l'échéance du 22/12/2022 pour la première vague ou pour l'échéance finale du 18/05/2023.

Notez que les procédures de remise du projet et d'évaluations sont dans le module « Projet ».

Le module projet correspond au module 7 de ce support de cours.

1.2.3 Les modalités d'évaluation de notre MOOC

NOTATION

Le cours comporte des évaluations de votre apprentissage tout au long des différents modules, soit sous forme de quiz soit sous forme d'exercices de codage. Aucune évaluation finale sous forme d'un examen ou quiz de fin de cours n'est donc proposée.

L'évaluation tout au long de l'apprentissage s'effectue par un système classique de points, et votre note sera calculée sur un total de **300 points**, répartis en :

— 102 points sur les **quiz** qui déterminent si vous avez assimilé la matière à raison de **1 point par réponse correcte**. **Attention vous n'avez généralement qu'un seul essai pour obtenir la bonne réponse** ;
— 150 points sur les **codes UpyLaB** à fournir qui valident votre autonomie dans le codage avec la matière vue jusque-là, à raison généralement de **2 points par exercice UpyLaB noté** qui passe correctement la vérification ;
— 48 points sur le **projet** associé au cours ; les points sont répartis en 24 points provenant des **évaluations par vos pairs** et 24 points provenant d'une **auto évaluation** du projet éventuellement amendé suite aux remarques de vos pairs.

Notez que quelques exercices UpyLaB ne sont pas notés même s'ils participent à votre apprentissage.

REMARQUES IMPORTANTES

1) Contrairement aux quiz, les exercices UpyLaB peuvent être testés autant de fois que vous le désirez.
2) Pour enregistrer votre réponse dans un exercice UpyLaB vous devez cliquer sur le bouton `Valider`, et cela, même si vous savez que votre code n'est pas correct ; ce n'est pas un souci puisque le nombre de fois que vous vérifiez chaque exercice UpyLaB n'est pas limité.
3) Attention, si, pour un exercice UpyLaB, la dernière vérification est négative, la plateforme ne retient pas si une vérification précédente avait été positive. En clair, la plateforme ne vous accorde les points que si la **dernière** vérification de cet exercice est positive.
4) Si vous décidez de ne pas faire le projet, pour obtenir votre attestation de réussite, vous devrez obtenir 150 points sur les 252 points qui restent en jeu.
5) De même ne pas faire certains exercices UpyLaB signifie que sciemment, vous décidez de ne pas avoir de points pour ces exercices. Vous pourrez toujours, avant l'échéance finale du cours, changer d'avis pour améliorer votre score.

ATTESTATION DE SUIVI AVEC SUCCÈS

Tout apprenant atteignant un taux de réussite de 50% des points avant l'échéance recevra une **attestation de suivi avec succès** produite et envoyée par la plateforme FUN.

1.2.4 Naviguer dans notre MOOC

COMMENT NAVIGUER DANS UN MOOC FUN

Un MOOC contient de nombreuses ressources vidéos, textuelles, des exercices à soumettre et à faire viser par la plateforme mais aussi des forums de discussions, des tableaux montrant votre progression, etc.

Il est donc important de bien maîtriser la plateforme FUN pour être à l'aise dans ces aspects pratiques. Pour les débutants sur FUN, un mini cours de démonstration de la plateforme FUN est proposé pour vous y familiariser.

Note : Voir le cours en ligne section 1.2.4 pour accéder au mini cours d'utilisation de la plateforme FUN.

RESSOURCES ET ÉLÉMENTS DE NAVIGATION DE NOTRE MOOC

Si nous parcourons notre MOOC « Apprendre à coder avec Python », les onglets suivants sont présents :

Cours

Il vous donne accès aux 7 modules du cours dont le module « Projet », au fur et à mesure de leur mise à disposition.

Infos du cours

L'équipe pédagogique y communique des nouvelles et informations importantes comme l'ouverture d'un module, un rappel des échéances, ou encore des éventuelles informations sur l'état d'avancement du cours.

Discussions

Vous y trouverez les forums ou espaces de discussion du cours :

- Forum pour faire connaissance en début de cours (module 1) ;
- Forums généraux de chaque module : pour répondre aux questions sur la matière du module ou aux soucis d'accès aux pages … ;
- Forums des exercices de codage UpyLaB : notre outil que nous présentons dans une section suivante et qui vous permettra de vous exercer ; ces forums sont à utiliser si vous avez des soucis avec un exercice de codage UpyLaB que vous n'arrivez pas à résoudre malgré la consultation du cours et de la FAQ (Foire Aux Questions) sur le sujet ;
- Forum sur le projet et son évaluation (réalisée au module « Projet ») ;

Les échanges entre les apprenants et l'équipe pédagogique se font majoritairement via ces forums.

Wiki

Il ne sera utilisé que pour déposer les oeuvres d'art que vous et les autres apprenants aurez produites grâce au programme réalisé en projet.

Progression

Il contient deux parties :

1) D'abord un tableau récapitulatif des notes obtenues jusqu'à présent, comme le montre la figure « Tableau récapitulatif de la progression » :

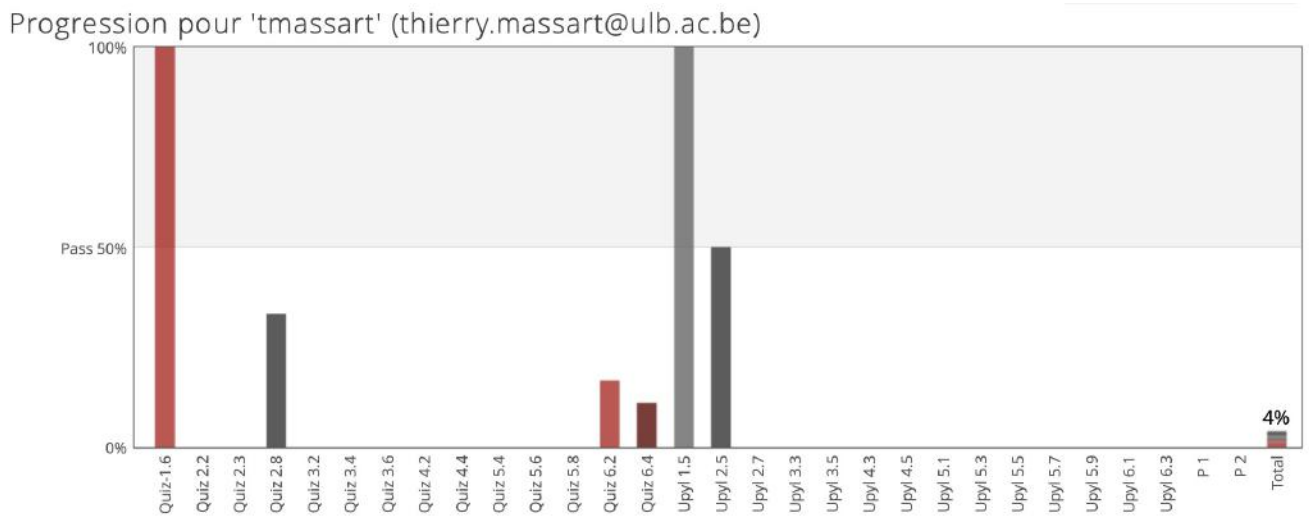
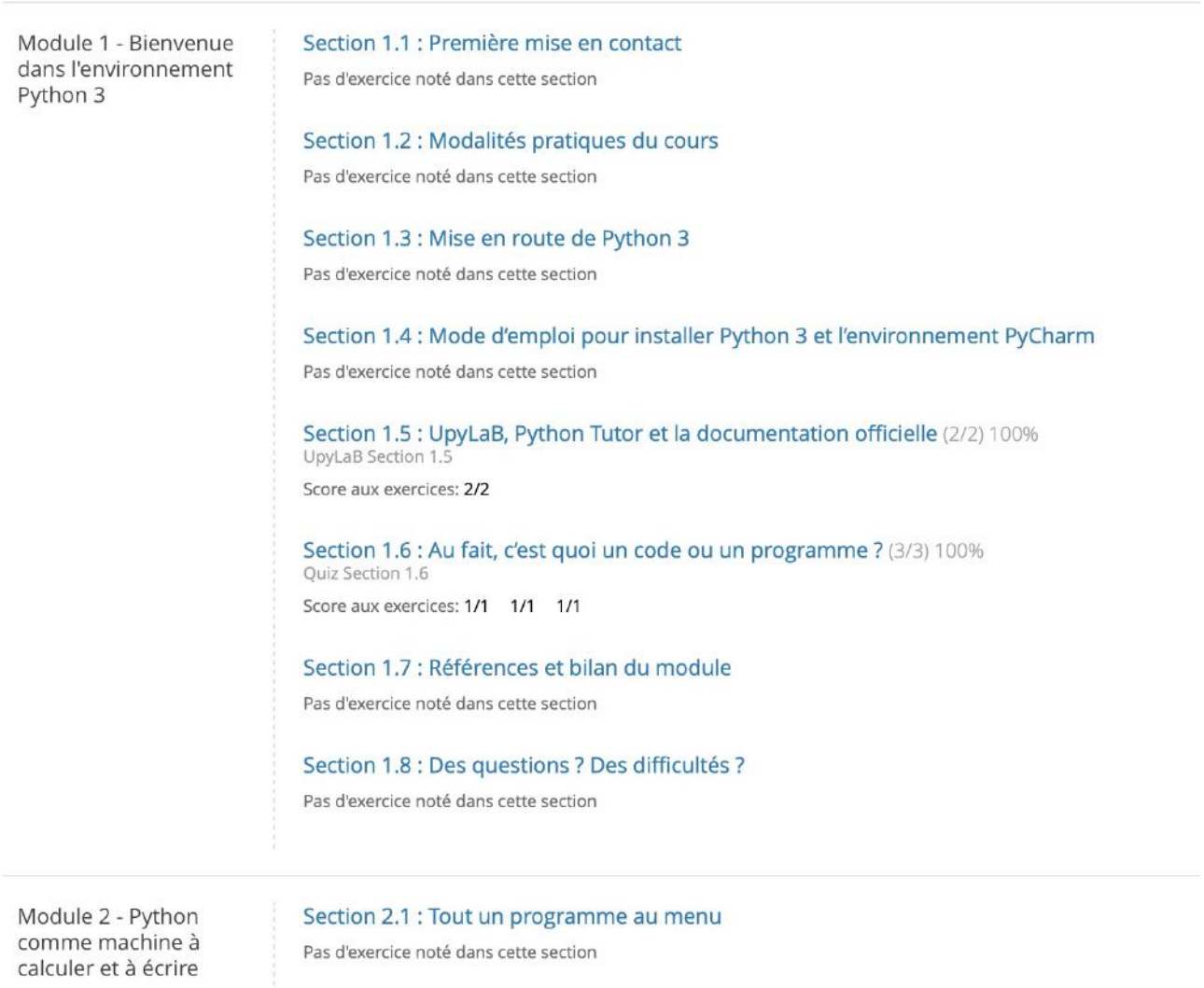


Tableau récapitulatif de la progression

Ce tableau contient :

- d'abord des colonnes se rapportant aux scores des quiz : avec autant de colonnes que de sections où un quiz est proposé ; par exemple Quiz 2.2 qui correspond aux quiz de la section 2.2 ;
- ensuite des colonnes se rapportant aux scores des exercices de codage UpyLaB, avec de même autant de colonnes que de sections où un ou des exercices de codage UpyLaB sont proposés ; par exemple : Upyl 2.5 qui correspond aux exercices UpyLaB de la section 2.5 ;
- deux colonnes pour les notes du projet (évaluation par les pairs et auto-évaluation) ;
- et enfin le pourcentage approximatif du *Total* obtenu.

2) Ensuite la liste des sections avec pour chaque section, les types d'exercices dans cette section et les notes possibles et obtenues jusqu'à présent, comme le montre la figure « Exercices dans chaque section » :


Module 1 - Bienvenue dans l'environnement Python 3
Section 1.1 : Première mise en contact
Pas d'exercice noté dans cette section
Section 1.2 : Modalités pratiques du cours
Pas d'exercice noté dans cette section
Section 1.3 : Mise en route de Python 3
Pas d'exercice noté dans cette section
Section 1.4 : Mode d'emploi pour installer Python 3 et l'environnement PyCharm
Pas d'exercice noté dans cette section
Section 1.5 : UpyLaB, Python Tutor et la documentation officielle (2/2) 100%
UpyLaB Section 1.5
Score aux exercices: 2/2
Section 1.6 : Au fait, c'est quoi un code ou un programme ? (3/3) 100%
Quiz Section 1.6
Score aux exercices: 1/1 1/1 1/1
Section 1.7 : Références et bilan du module
Pas d'exercice noté dans cette section
Section 1.8 : Des questions ? Des difficultés ?
Pas d'exercice noté dans cette section
Module 2 - Python comme machine à calculer et à écrire
Section 2.1 : Tout un programme au menu
Pas d'exercice noté dans cette section

Exercices dans chaque section

1.2.5 Aperçu de la suite de ce module

APERÇU DE LA SUITE DE CE MODULE

Note : Voir la vidéo de la section 1.2.5 du cours en ligne : Installation de Python 3

CONTENU DE LA VIDÉO

La vidéo précédente présente la suite du module 1 : l'installation de l'interpréteur Python 3 ainsi que de l'environnement de développement PyCharm qui seront utilisés tout au long de ce cours.

Remarque :

Notez que pour les sessions précédentes de ce MOOC, comme présenté dans la vidéo qui suit, nous avions conseillé d'utiliser l'environnement PyCharm Community, qui permet de réaliser beaucoup d'aides aux programmeurs et est gratuit, même s'il a été conçu par une société privée (Jetbrains). A partir de cette session, nous vous proposons d'installer Thonny : un environnement de développement plus simple mais qui fait parfaitement l'affaire. En effet, PyCharm est trop complexe pour un débutant en programmation par rapport aux bénéfices qu'il va en tirer.

Dans les vidéos que nous présentons, c'est l'environnement PyCharm qui est utilisé ; si vous utilisez Thonny, la correspondance entre PyCharm et Thonny ne devrait pas vous poser de problèmes.

1.3 Mode d'emploi pour installer un environnement de développement

1.3.1 Que va-t-on installer ?

Deux « outils » sont indispensables à votre apprentissage :

— un **interprète Python3** qui va vous permettre d'exécuter vos codes sur votre ordinateur
— un **éditeur de texte** pour saisir et enregistrer vos codes

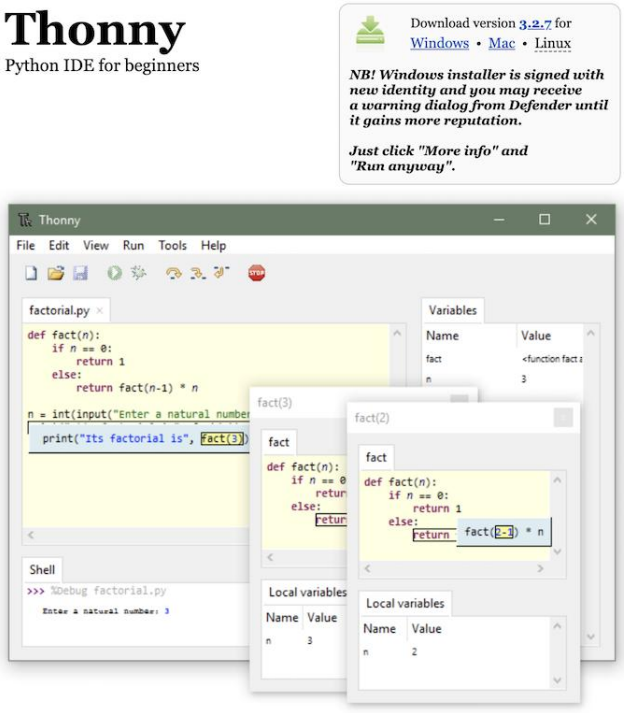
Les deux outils sont indépendants et peuvent être installés séparément. C'est l'option que nous avions choisie dans les premières sessions du MOOC. Les pages suivantes dans cette section détaillent donc l'installation de l'interprète Python3 et de l'éditeur de texte PyCharm sur les trois systèmes d'exploitation (Windows, MacOs et Linux). Dans l'ensemble des vidéos, c'est PyCharm qui est utilisé.

PyCharm est plus qu'un simple éditeur de texte. Il s'agit en réalité d'un Environnement de Développement Intégré (EDI ou IDE pour l'acronyme anglais) qui permet non seulement d'éditer ses codes mais aussi de les exécuter. L'installation de cet EDI est détaillée ainsi que son utilisation de base dans les pages suivantes.

Toutefois, la configuration de PyCharm peut s'avérer difficile dans certains cas et c'est pour cela que nous proposons une alternative, a priori plus simple : l'EDI Thonny. Comme PyCharm, Thonny est outil qui sert à la fois à écrire ses codes Python et à les exécuter. Tout ce qui vous est montré dans les vidéos en utilisant PyCharm peut par conséquent être reproduit avec Thonny.

INSTALLATION DE L'IDE THONNY

La méthode la plus simple pour installer Thonny consiste à se rendre sur le site de l'application https://thonny.org/ et à télécharger la dernière version. L'installation se fait par simple clic, comme la plupart des applications.


Thonny
Python IDE for beginners
Download version 3.2.7 for
Windows • Mac • Linux
NB! Windows installer is signed with new identity and you may receive a warning dialog from Defender until it gains more reputation.
Just click "More info" and "Run anyway".
Thonny
File Edit View Run Tools Help
factorial.py
Variables
Name Value
Shell
Local variables

Le principal avantage de cet IDE est qu'il embarque aussi un interprète Python. Cela signifie que vous n'avez rien d'autre à installer ; ce qui est particulièrement intéressant pour les utilisateurs du système Windows qui ne possède pas forcément d'interprète Python installé par défaut (contrairement aux systèmes MacOS et Linux).

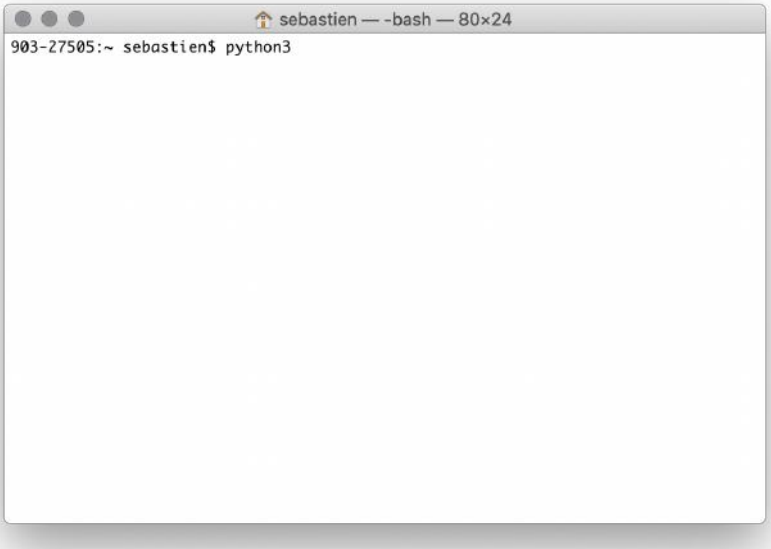
Si vous avez réussi à installer Thonny et à le lancer, vous pouvez passer directement à la section suivante montrant l'utilisation de l'IDE.

Pour les autres, et notamment ceux ou celles qui travaillent avec les systèmes MacOs ou Linux, voici une autre méthode d'installation et de lancement de cette application.

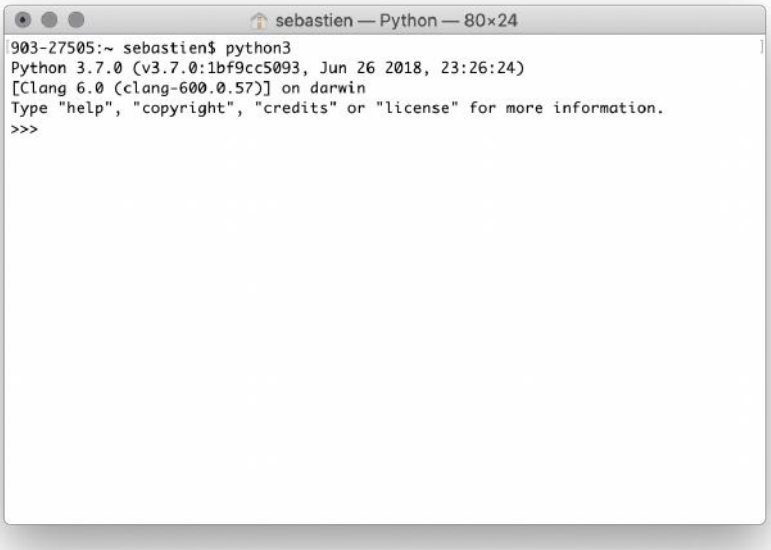
Vous avez normalement Python déjà installé sur vos machines. Pour vous en rendre compte :

— Recherchez sur votre ordinateur l'application Terminal puis exécutez-la. Vous devriez avoir l'ouverture d'une fenêtre très basique (avec des noms différents pour la machine et l'utilisateur bien sûr) :

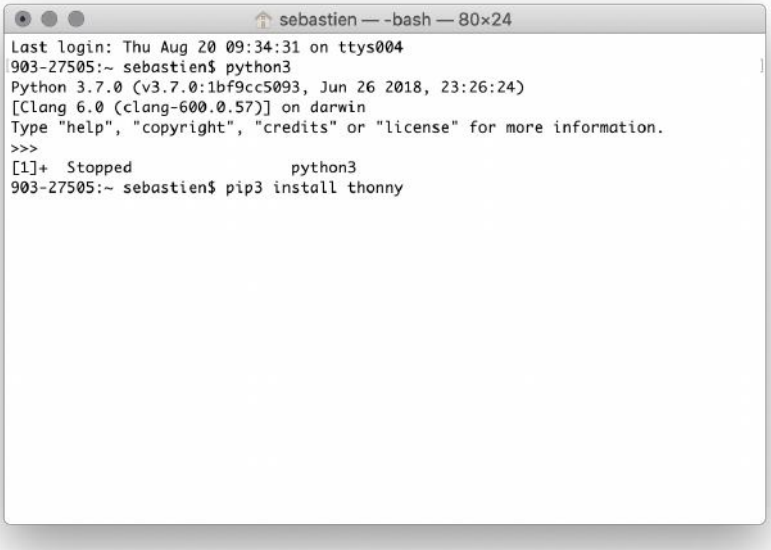
— Exécutez la commande *python3* (ou python si cela ne fonctionne pas) :



— Une fois lancé, l'interprète Python devrait s'afficher (vous le reconnaissez au triple symbole >) :

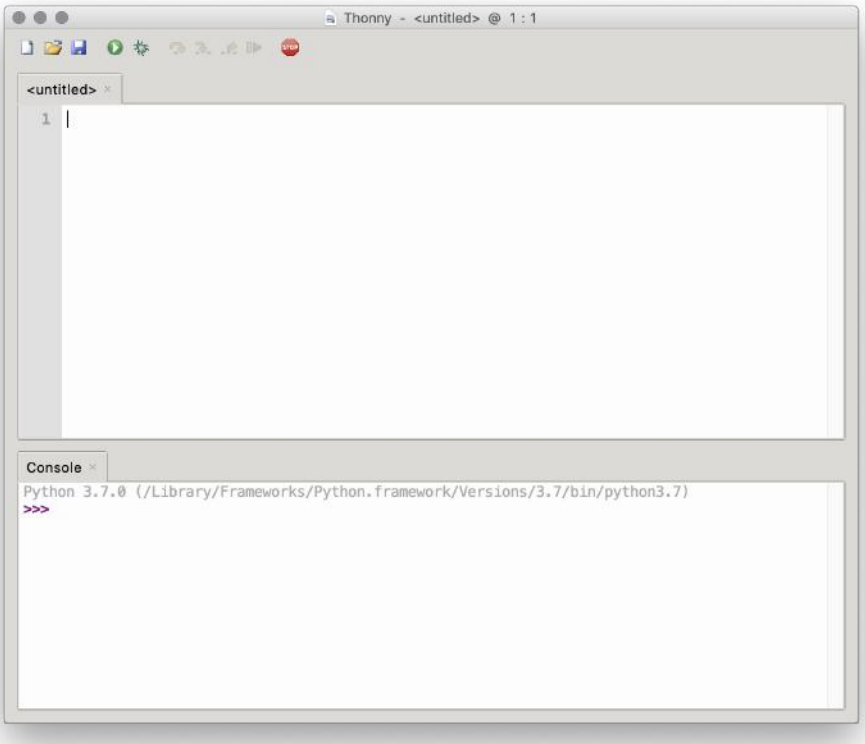

sebastien — Python — 80×24
903-27505:~ sebastien$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

— La première ligne vous renseigne sur la version de votre interprète (ici 3.7.0). Si cette version est supérieure ou égale à 3.4 alors l'utilitaire *pip* est installé. Vous pouvez quitter votre interprète Python3 (un simple CTRL-Z). Assurez-vous d'être connecté à internet puis exécutez la commande : *pip3 install thonny* (ou *pip install thonny* si cela ne fonctionne pas ou encore *sudo apt install thonny*). Pour lancer l'IDE il suffit alors de taper la commande : *thonny*


sebastien — -bash — 80×24
Last login: Thu Aug 20 09:34:31 on ttys004
903-27505:~ sebastien$ python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
[1]+ Stopped python3
903-27505:~ sebastien$ pip3 install thonny

1.3.2 Utilisation de Thonny

Vous avez installé Thonny sur votre ordinateur. Lancez-le, vous devriez avoir une fenêtre similaire à :



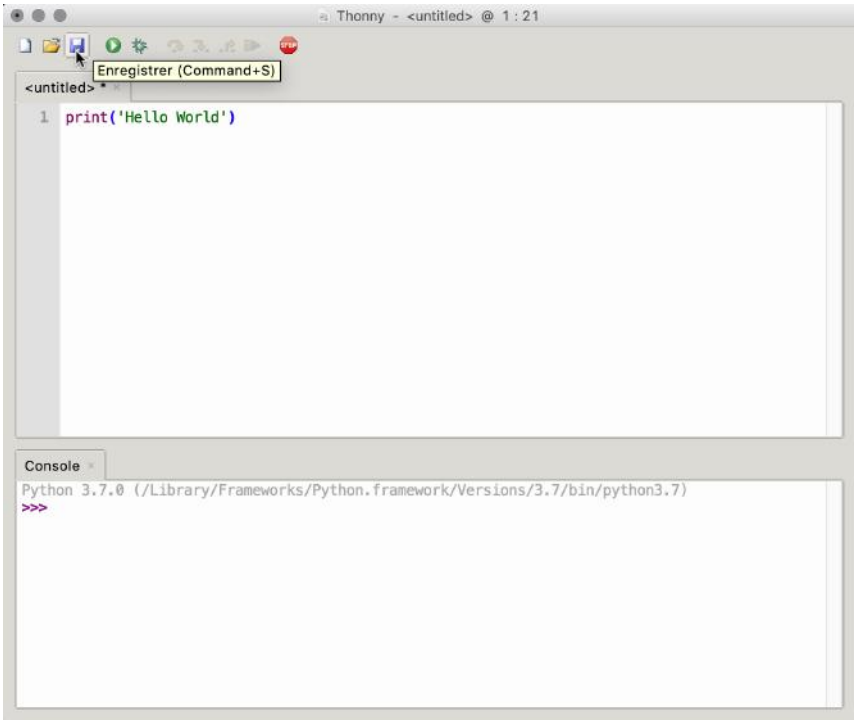
Cette fenêtre comporte deux zones :

— la partie supérieure est la partie **éditeur de texte** pour la saisie de vos codes

— la partie inférieure est l'"**interprète Python** où vous pouvez taper des instructions python. C'est dans cette zone que vous verrez vos codes s'exécuter.

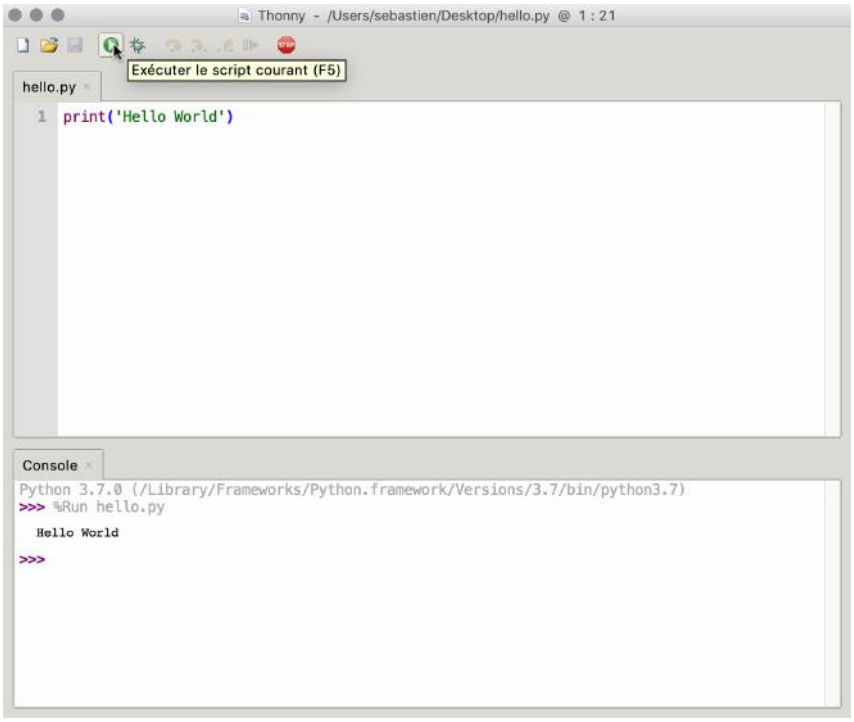
Voici quelques étapes de l'utilisation de Thonny.

— Édition d'un premier code (appelé script)



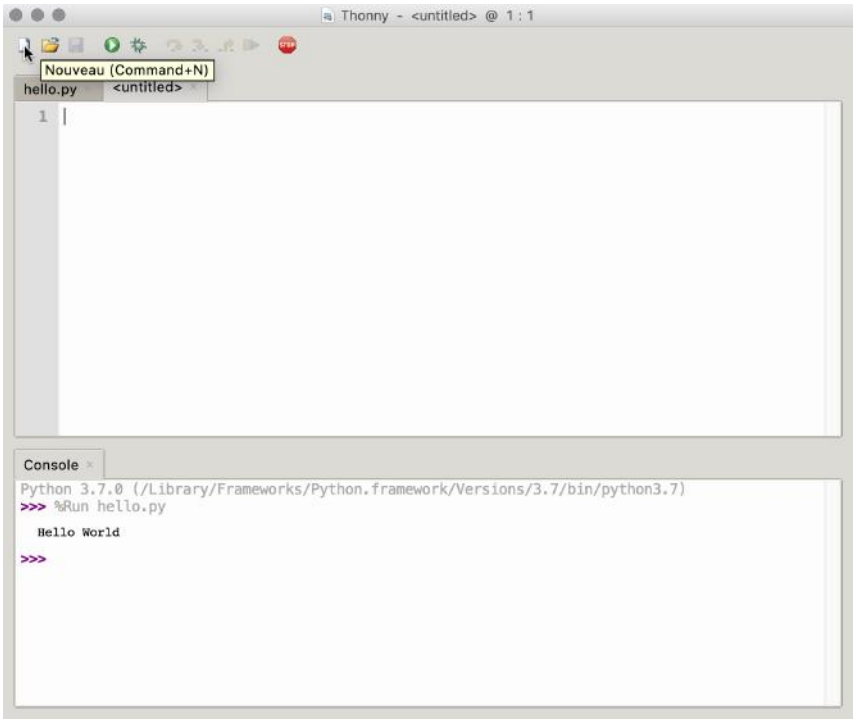
Dès que vous avez cliquez sur le bouton *Enregistrer*, vous naviguez dans votre système de fichier de façon classique pour l'enregistrement d'un document (en créant éventuellement un nouveau dossier).

— Pour exécuter le code il suffit de cliquer sur la flèche verte :

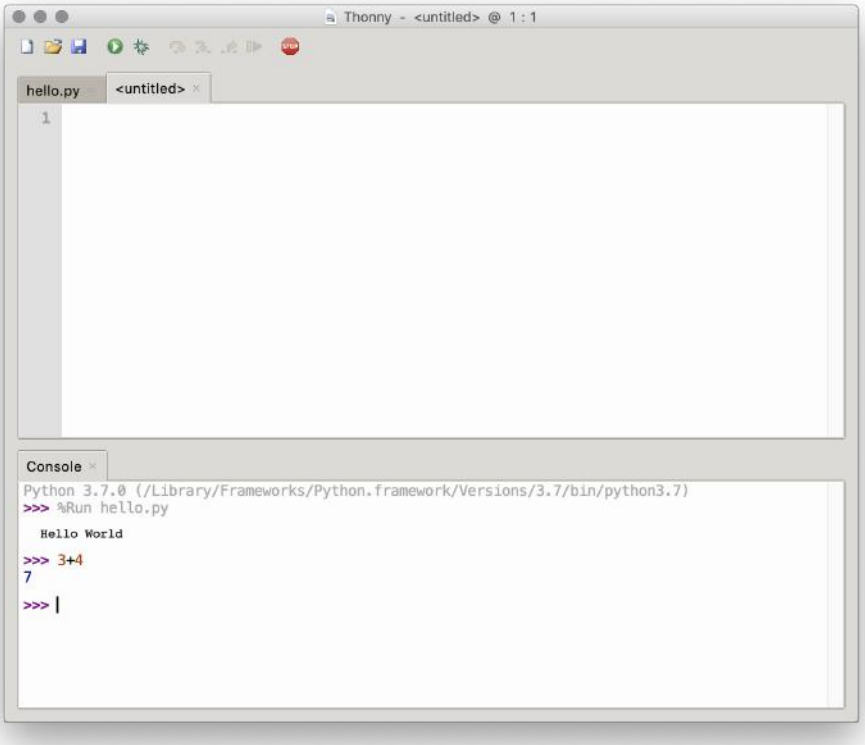


Notez le *Hello World* qui s'est affiché dans la partie interprète.

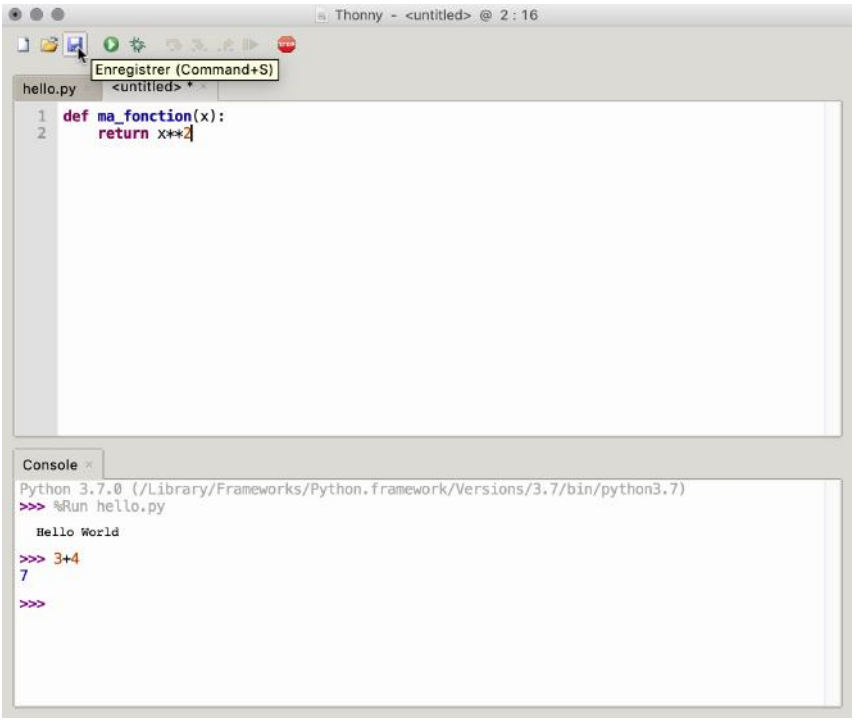
— On peut ouvrir une nouvelle zone éditeur vierge :



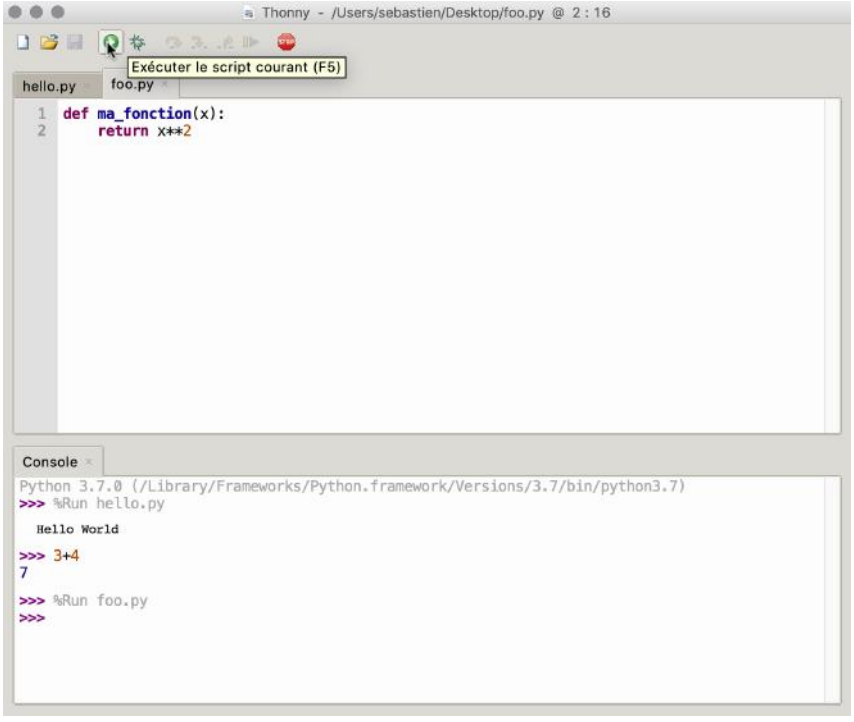
— On peut se servir de l'interprète pour tester des expressions Python :



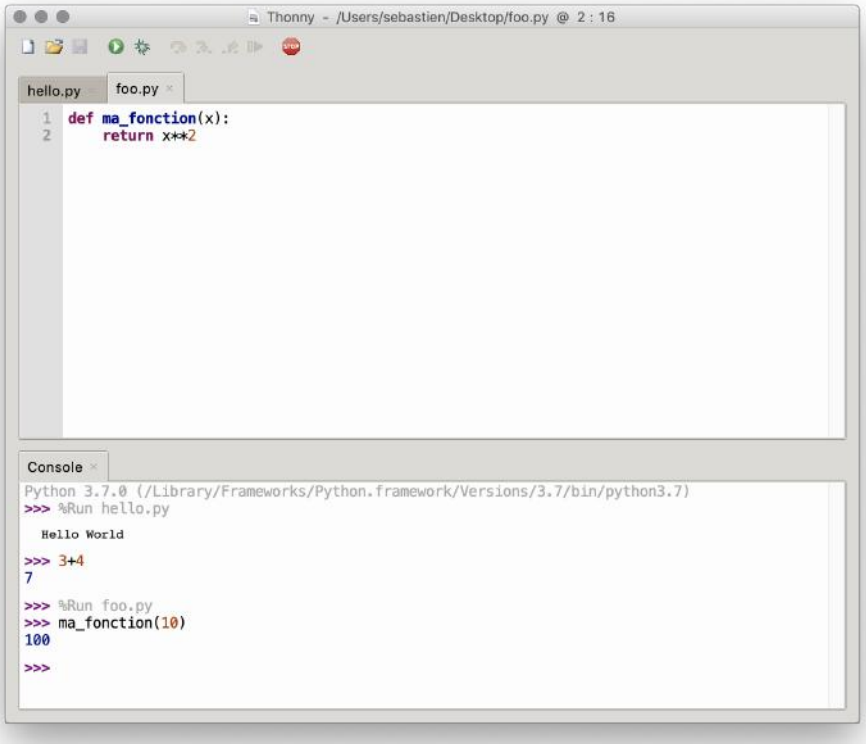
— Ou même des fonctions :



Ici l'exécution ne produit aucune trace puisque notre script ne comporte que la définition d'une fonction (vous verrez en détail ces notions, ne vous inquiétez pas). Toutefois, l'indication *%Run foo.py* dans la partie interprète nous dit que le code a bien été exécuté :



Nous pouvons alors tester notre fonction :



Les quelques exemples présentés couvrent l'utilisation de base de Thonny, utilisation suffisante dans le cadre du MOOC. Pour une utilisation avancée, notamment du déboggueur, nous vous invitons à consulter la documentation officielle : https://thonny.org/

1.3.3 Si je veux quand même installer PyCharm plutôt que Thonny sur mon ordinateur Windows

SUR MON ORDINATEUR WINDOWS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3 - EN RÉSUMÉ

Ouvrez un navigateur Web et allez sur le site officiel de Python. Cliquez sur le menu Download -> Python3.x.x (Python 3.6.3 ou ultérieur). Si le site ne propose pas spontanément la bonne version, vous devez sélectionner celle qui convient au système de votre ordinateur. L'interpréteur Python 3 s'installe en cliquant sur « exécuter » du programme d'installation.

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR WINDOWS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3

Note : Voir la première vidéo de la section 1.3.2 du cours en ligne : Installation de Python 3 sur un ordinateur Windows

SUR MON ORDINATEUR WINDOWS

COMMENT INSTALLER PYCHARM - EN RÉSUMÉ

— Ouvrons un navigateur web sur le page : https://www.jetbrains.com/pycharm/
— Cliquons sur le bouton « Download » qui propose de télécharger PyCharm pour votre système, Windows, MacOs ou Linux dans l'édition Professionnelle ou Community.
— C'est la version Community qu'il nous faut prendre en cliquant sur le bouton download idoine.
— Nous autorisons que le package d'installation (fichier avec le suffixe .exe) s'exécute sur notre ordinateur.
— Nous procédons à son installation en autorisant son exécution et répondant aux questions qu'il pose (y compris si c'est une version 32 ou 64 et si les fichiers .py sont associés à PyCharm - sélectionnez l'option).
— Un petit conseil : installons l'icône PyCharm sur le bureau pour pouvoir ouvrir facilement l'environnement (après l'installation elle se trouve dans les applications JetBrains).

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR WINDOWS

INSTALLER PYCHARM

Note : Voir la seconde vidéo de la section 1.3.2 : Installation de l'environnement PyCharm sur un ordinateur Windows

---

1.3.4 Si je veux quand même installer PyCharm plutôt que Thonny sur mon ordinateur MacOs

SUR MON ORDINATEUR MACOS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3 - EN RÉSUMÉ

Ouvrez un navigateur Web et allez sur le site officiel de Python. Cliquez sur le menu Download -> Python3.x.x (Python 3.6.3 ou ultérieur). Si le site ne propose pas spontanément la bonne version, vous devez sélectionner celle qui convient au système de votre ordinateur. L'interpréteur Python 3 s'installe en cliquant sur « exécuter » du programme d'installation.

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR MACOS

COMMENT INSTALLER L'INTERPRÉTEUR PYTHON 3

Note : Voir la première vidéo de la section 1.3.3 : Installation de Python 3 sur un ordinateur MacOS

SUR MON ORDINATEUR MACOS

COMMENT INSTALLER PYCHARM - EN RÉSUMÉ

— Ouvrons un navigateur web sur la page : https://www.jetbrains.com/fr-fr/pycharm/
— Cliquons sur le bouton « Download » qui propose de télécharger PyCharm pour votre système, Windows, MacOs ou Linux dans l'édition Professionnelle ou Community
— C'est la version Community qu'il nous faut prendre en cliquant sur le bouton download idoine.
— Nous attendons que le package d'installation soit téléchargé sur notre ordinateur. C'est un fichier avec le suffixe .dmg (ce serait un exe avec le système windows).
— Ensuite procédons à son installation en ouvrant le package d'installation et en plaçant l'application dans le répertoire des applications.

La vidéo suivante montre tout ceci plus en détails.

SUR MON ORDINATEUR MACOS

INSTALLER PYCHARM

Note : Voir la seconde vidéo de la section 1.3.3 : Installation de PyCharm sur un ordinateur MacOS

1.3.5 Si je veux quand même installer PyCharm plutôt que Thonny sur mon ordinateur Ubuntu ou Linux

COMMENT INSTALLER PYTHON 3 ET PYCHARM SUR UBUNTU - EN RÉSUMÉ

Nous allons procéder à l'installation qui vous permettra d'avoir l'interpréteur Python 3 et l'environnement PyCharm. Cette installation est décrite pour la distribution 16.04 du système d'exploitation Ubuntu ; une installation similaire peut être réalisée avec un ordinateur fonctionnant avec un autre système Linux.

Python 3 est déjà installé dans cette distribution, excepté le module turtle qui sera utilisé lors du cours. Pour ajouter ce module utilisez la commande :

```
sudo apt-get install python3-tk
```

Par contre PyCharm n'est pas installé.

COMMENT INSTALLER PYCHARM SUR MON ORDINATEUR UBUNTU - RÉSUMÉ

La procédure est disponible via le site http://ubuntuhandbook.org et une recherche sur le mot clé PyCharm.

Ouvrez un terminal (par exemple via Ctrl+Alt+T ou en cherchant et lançant l'application). Dans le terminal exécutez la commande :

```
sudo snap install pycharm-community --classic
```

sudo demande le mot de passe, que vous devez introduire terminé par la touche Enter. Normalement l'installation s'effectue.

La vidéo suivante montre la procédure d'installation et de mise en route de PyCharm.

COMMENT INSTALLER PYCHARM SUR UBUNTU

Note : Voir la vidéo de la section 1.3.4 : Installation de PyCharm sur un ordinateur Linux

1.3.6 Si l'installation ne fonctionne pas

ET SI L'INSTALLATION NE FONCTIONNE PAS

Soit Python 3 et IDLE ont été installés sur votre ordinateur mais ce dernier n'a pas la configuration minimale pour l'installation de PyCharm : dans ce cas,

- soit installez Thonny : un éditeur simple mais parfaitement adapté à ce cours
- soit rabattez-vous sur l'IDE plus simple IDLE (fourni avec python3, mais pas toujours très stable)
- soit utilisez un éditeur simple pour écrire vos scripts et ensuite exécutez-les avec la commande qui invoque l'interpréteur (python3 sur linux ou macOs ou py sur windows) ;
- ou encore exécutez vos codes via le site trinket (voir plus bas).

Soit vous n'avez pas réussi non plus à installer Python 3. Dans ce cas exécutez vos codes via le site Trinket (voir ci-dessous).

Si la configuration de votre ordinateur est a priori suffisante : essayez de voir si le sujet existe dans la FAQ ou sinon mettez à profit l'entraide sur le forum du module, en décrivant précisément le souci et en demandant aux autres inscrits de vous aider. Si malgré tout, cela ne fonctionne pas, utilisez IDLE ou Trinket.

ENTRAIDE LORS DE L'INSTALLATION DE PYTHON 3

Notre cours en ligne est aussi un endroit (virtuel) où l'entraide est la règle d'or. Le forum du module, en fin de module, permet de mettre en contact les apprenants (voir règles de bonnes pratiques des forums via le lien https://www.fun-mooc.fr/asset-v1:ulb+44013+session06+type@asset+block@BonnesPratiquesForum.pdf).

En particulier dans l'activité précédente :

- Si vous avez rencontré des soucis dans l'installation de l'environnement Python, décrivez le plus clairement possible le problème et précisez **surtout** le système d'exploitation qu'utilise votre ordinateur (exemple : Windows 10 version 1703 ou MacOS 10.12.4…) : notre équipe ou d'autres apprenants qui ont la solution pourront probablement vous aider.
- Si vous avez réussi l'installation après avoir résolu quelques soucis, merci de partager votre expérience en détaillant le système d'exploitation utilisé par votre ordinateur, le souci rencontré et sa résolution.
- Si vous avez la solution à un problème évoqué dans le forum, aider ceux qui l'évoquent en leur fournissant une réponse leur sera d'un grand secours et fera avancer la communauté formée par l'ensemble des participants de notre cours en ligne.

TRINKET SI VOUS N'AVEZ PAS RÉUSSI À INSTALLER PYTHON 3

Trinket permet d'exécuter du code Python sur une plateforme distante, c'est-à-dire sans que l'interpréteur Python ou un IDE local soit installé sur son ordinateur.

Trinket est une bonne alternative pour apprendre Python 3 même si vous n'aurez pas la flexibilité offerte par un environnement complet comme fourni par PyCharm.

Il suffit de se connecter à l'adresse https://trinket.io et de créer un compte en cliquant sur l'onglet Sign up et en remplissant le formulaire (votre nom, un nom d'utilisateur, votre adresse email et un mot de passe).

Une fois votre compte créé, vous pouvez vous connecter et trouver l'interpréteur Python 3 par exemple en cliquant sur le bouton new trinket -> Python 3.

Vous pouvez dès lors taper votre code dans la fenêtre main.py : la première ligne du code doit toujours être la suivante :

```
#!/bin/python3
```

qui indique à Trinket que la suite est du Python 3 (avec la convention Unix).

Par exemple :

```
#!/bin/python3
print('Hello World')
```

et ensuite vous pouvez demander l'exécution du code en cliquant sur le triangle au dessus de la fenêtre main.py

1.3.7 Installation d'un dictionnaire français dans PyCharm

L'environnement de développement PyCharm est d'une grande aide aux codeurs. En particulier, il vérifie si ce que nous encodons semble être du code Python cohérent. Il nous mentionne également si nous utilisons dans nos explications (nous verrons qu'elles s'appellent commentaires) des mots corrects. Pour cela PyCharm utilise un ou des dictionnaires.

Pour nous aider, nous proposons d'installer le dictionnaire des mots français (y compris, nous verrons pourquoi, les mots dont les accents ont été supprimés). Avant de procéder à l'installation elle-même, nous devons télécharger ces deux dictionnaires.

TÉLÉCHARGEMENT DES FICHIERS

Vous pouvez réaliser ce téléchargement :
- dictionnaire_mots_français via le lien mots
- dictionnaire_français_avec_et_sans_accents via le lien mots_avec_et_sans_accents

Maintenant que les dictionnaires sont téléchargés sur votre ordinateur (assurez-vous de savoir où ils se trouvent), renommez les `mots.dic` et `mots_avec_et_sans_accents.dic` et voyons la procédure pour installer ces dictionnaires dans l'environnement PyCharm.

INSTALLATION DANS PYCHARM

- Après avoir ouvert PyCharm, cliquez sur le menu `File` ensuite `Settings` (ou `PyCharm` ensuite `Preferences` suivant le système que vous utilisez (Windows, MacOS, ...) (voir figure « Installation dictionnaire étape 1 ») ;

- cliquez sur le petit triangle devant l'intitulé `Editor` pour ouvrir ce sous-menu (voir figure « Installation dictionnaire étape 2 ») ;

- ensuite cliquez sur `spelling` (voir figure « Installation dictionnaire étape 3 ») ;

- dans la sous-fenêtre `Custom Dictionaries` cliquez sur le bouton + (voir figure « Installation dictionnaire étape 4 ») ;
- et trouvez et sélectionnez le dictionnaire à ajouter en cliquant sur `open` ;

- le dictionnaire s'ajoute à la liste des dictionnaires (voyez les deux nouvelles coches) (voir figure « Installation dictionnaire étape 5 ») :

- sortez du menu en cliquant sur le bouton `ok`.

UTILISER PYCHARM

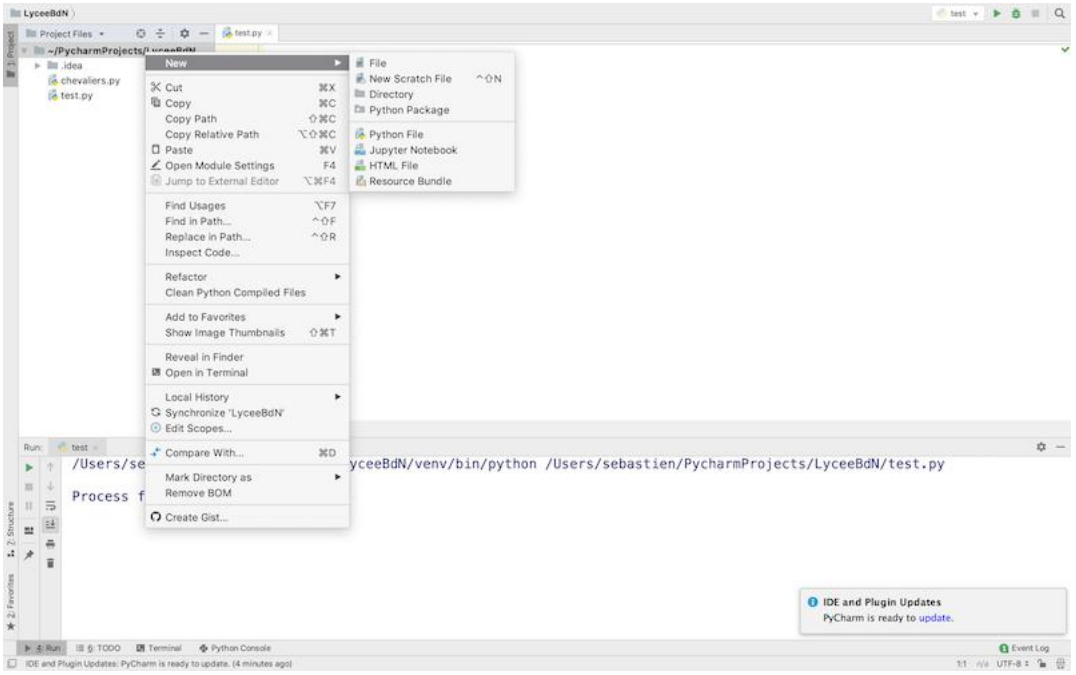
1.3.8 Utilisation de PyCharm

Vous avez installé PyCharm. Lancez-le et créez un nouveau projet. Voyons comment créer et exécuter un script.

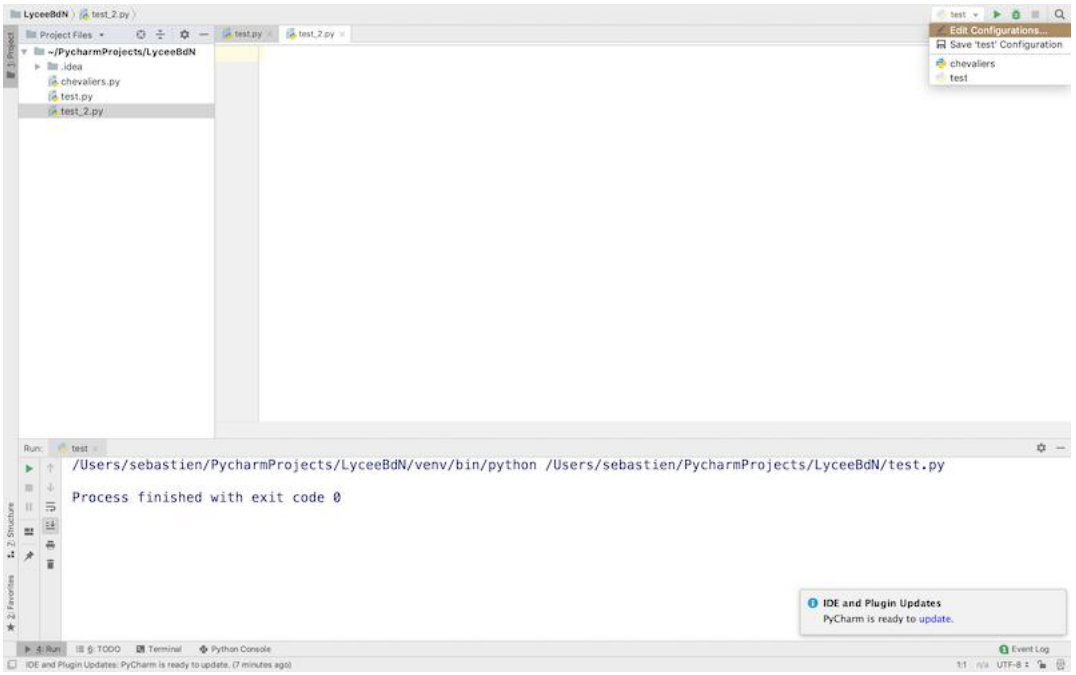
Un projet peut contenir plusieurs scripts bien sûr. Soit qui sont liés (c'est le cas dans les gros projets réels) soit qui sont indépendants mais appartiennent à une même famille… C'est le cas par exemple si je décide de faire un projet par Module de mon MOOC : je mettrai dans le projet `Module_1` tous mes exercices du Module 1, etc. On peut aussi faire un gros projet `MOOC_Python` et y mettre tous ses scripts sans distinction de module.

Cette organisation est bien sûr question de goût, d'habitude de travail et donc propre à chacun.

Lorsque votre projet est créé, pour y ajouter un nouveau script vous pouvez utiliser un clic droit sur le nom du projet, dans la colonne de gauche, et faire `New > Python File`. Vous donnez un nom à ce script qui vient se rajouter aux autres. Dans la colonne de gauche vous avez l'ensemble des scripts. Ainsi, les deux captures suivantes montrent comment j'ai créé un nouveau script nommé `test_2.py` (voir les deux figures « Création d'un nouveau script dans PyCharm »).



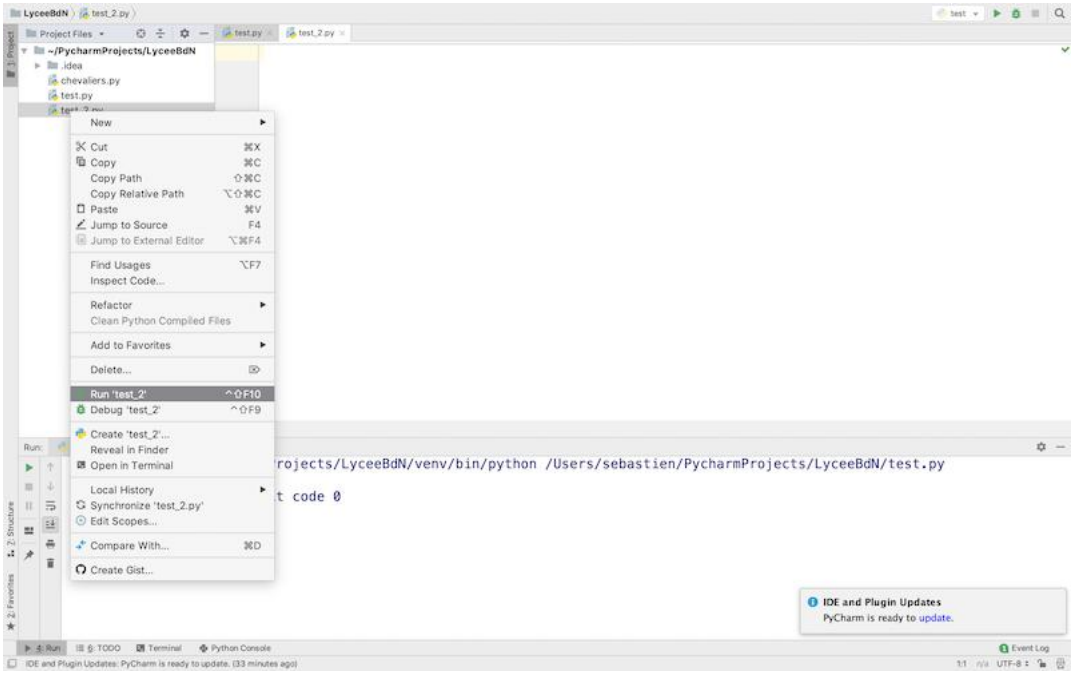
Création d'un nouveau script dans PyCharm



Création d'un nouveau script dans PyCharm (suite)

Parmi tous les scripts de votre projet, pour en exécuter un, il y a deux cas de figure :

1) soit il est sélectionné comme script courant (il apparaît en haut à droite avec la petite flèche verte) auquel cas utiliser cette petite flèche ou faire `Run` dans le menu exécutera ce script ;
2) soit il ne l'est pas et alors le plus simple est de faire, dans la colonne de gauche, un clic droit sur le nom du script que vous voulez exécuter (disons `test_2.py` comme sur la capture ci-dessous) et choisir dans le menu contextuel `Run test_2` (voir figure « Exécution d'un script test_2 dans PyCharm ») :

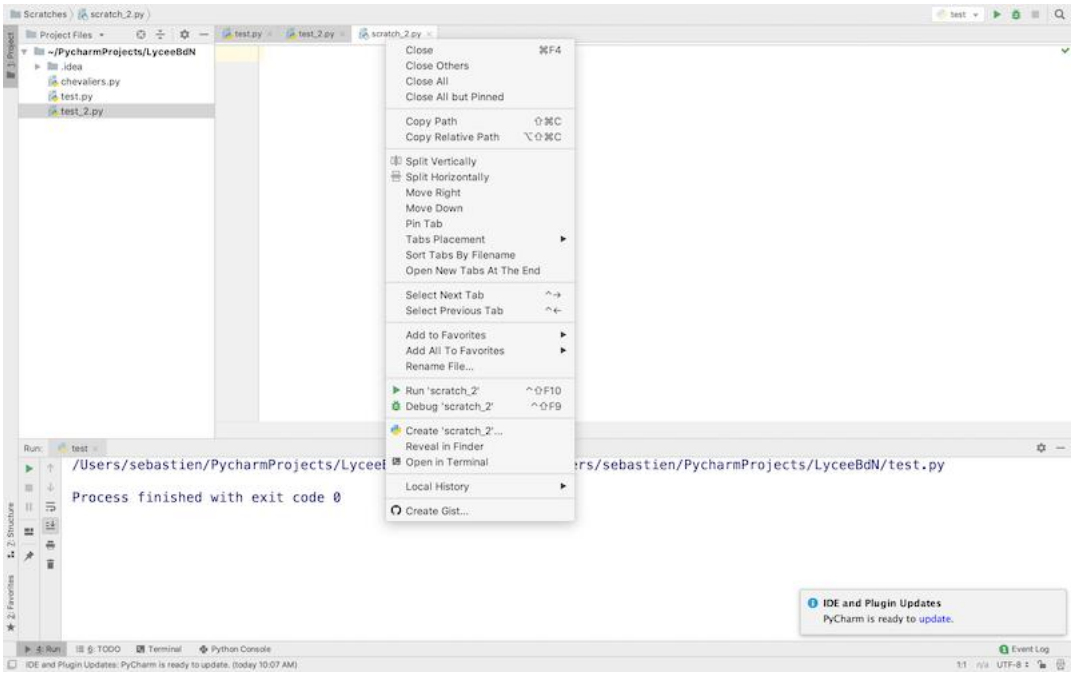


Exécution d'un script test_2 dans PyCharm

Et les scratch file ?

Les scratch files sont des fichiers python comme les autres mais numérotés automatiquement et rangés dans un dossier (projet) à part. Par exemple ci-dessous j'ai créé un nouveau scratch file, (ce n'est pas mon premier comme l'atteste le 2 dans le nom) et il

n'apparaît pas dans mon Projet (qu'on voit à gauche). Mais si je fais apparaître le menu contextuel en cliquant-droit sur le nom `scratch_2.py`, je vois que j'ai un `Run scratch_2` et je peux donc l'exécuter (voir figure « Exécution d'un script dans un fichier scratch dans PyCharm »).



Exécution d'un script dans un fichier scratch dans PyCharm

Conseil

De préférence, créez des fichiers python dans votre ou vos projets et n'utilisez pas les *scratch files* de peur de ne plus savoir où se trouvent vos codes.

1.4 UpyLaB, Python Tutor et la documentation officielle

1.4.1 Notre exerciseur UpyLaB

UPYLAB

Dans ce cours, en plus de l'interpréteur Python 3 et de l'environnement de développement PyCharm, nous utilisons deux « outils » : **UpyLaB** et **Python Tutor**. Présentons d'abord UpyLaB.

UpyLaB est notre plateforme d'apprentissage en ligne. Elle propose des exercices de codage et demande aux étudiants de produire le code Python correspondant ; ensuite UpyLaB, qui est exécuté sur un serveur extérieur à la plateforme FUN, teste si la solution proposée est correcte. Nous avons intégré l'utilisation d'UpyLaB au cours en ligne, comme le montre le premier exercice UpyLaB proposé à la page suivante.

MODE D'EMPLOI

L'exercice UpyLaB 1.1 ci-après vous permet de tester l'utilisation d'UpyLaB dans le cours.

Le principe consiste à lire l'énoncé de l'exercice concerné puis à résoudre le problème votre environnement de développement (PyCharm ou Thonny par exemple). Une fois que vous vous êtes assuré que votre script répond aux exigences de l'énoncé, il faut le **recopier** dans la partie de fenêtre UpyLaB servant à cet effet (mini-fenêtre vide au départ, sinon qui contient votre dernière proposition de solution) pour le soumettre à évaluation en cliquant sur le bouton « Valider ».

Quand l'utilisateur a cliqué sur le bouton « Valider », UpyLaB va demander à l'interpréteur Python 3 d'exécuter le code fourni. Il va ensuite comparer le résultat obtenu avec ce qui est attendu et le valider si les deux coïncident.

UpyLaB effectue un ou plusieurs tests selon les exercices, et ensuite, en fonction des résultats de ces tests, vous indique si le code soumis lui semble correct ou non par rapport à ce qui vous est demandé.

Comme pour l'installation de Python 3, en cas de besoin, nous vous proposons d'utiliser la FAQ et si vos soucis ne sont pas résolus, le forum UpyLaB du présent module afin de vous mettre en contact avec les autres apprenants et l'animateur des forum.

En particulier, si vous rencontrez des soucis pour réaliser l'exercice 1.1 d'UpyLaB, décrivez le plus clairement possible le problème : notre équipe ou d'autres apprenants plus expérimentés pourront probablement vous aider.

1.4.2 Exercice UpyLaB 1.1 - Vert

TESTER UPYLAB EN FAISANT L'EXERCICE UPYLAB 1.1

Les seuls buts de cet exercice sont de vérifier que vous avez accès à notre exerciseur UpyLaB et de vous donner un premier contact avec cet outil.

Écrire un programme qui affiche « Bonjour UpyLaB ! » grâce à l'instruction :

```
print("Bonjour UpyLaB !")
```

Veillez à ne pas ajouter d'espace au début de la ligne et à parfaitement respecter les espaces, minuscules et les majuscules (ici pourquoi ne pas simplement faire une copier/coller de l'instruction dans la fenêtre UpyLaB)

Pour tous les exercices UpyLaB, vous devez écrire **le code Python qui solutionne le problème (c'est-à-dire votre programme Python)** dans la sous-fenêtre intitulée « Entrez votre solution ».

Exemple

Lorsqu'il est exécuté, le programme affiche :

```
Bonjour UpyLaB !
```

Bien sûr, libre à vous d'être curieux et :

— de voir ce que répond UpyLaB quand vous donnez une autre réponse, par exemple :

```
print("Bonsoir UpyLaB !")
```

ou en ajoutant une ou plusieurs espaces devant l'instruction ;

— même si l'instruction `print` ne vous a pas encore été présentée, de comprendre sa traduction française pour avoir une idée de ce qu'elle pourrait bien avoir comme effet.

Note : en cas de besoin, la section *PROCÉDURE POUR UTILISER UPYLAB* ci-dessous peut solutionner vos problèmes ou répondre à plusieurs de vos questions.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Vous obtenez une erreur de syntaxe ? : Vérifiez que vous avez bien copié l'instruction demandée, sans ajouter d'espace devant, et sans oublier parenthèses ou guillemets fermants.
- Pas d'erreur de syntaxe, mais UpyLaB refuse de valider votre code ? : L'affichage doit être exactement identique à celui attendu. Veillez en particulier à respecter majuscules / minuscules, à ne pas ajouter ou ôter des espaces (en particulier à la fin de la phrase), à ne pas oublier la ponctuation.
- Un conseil pour avoir un message identique est de le copier depuis l'énoncé pour le coller dans votre code. Ici, vous pouvez même copier toute l'instruction :

```
print("Bonjour UpyLaB !")
```

  avant de la coller dans la fenêtre UpyLaB de l'exercice.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 1.1*.

PROCÉDURE POUR UTILISER UPYLAB

Comme expliqué, chaque fenêtre UpyLaB donne un exercice à résoudre sous forme de code Python. Une procédure d'identification est mise en place entre les pages FUN et UpyLaB. Voici la procédure précise qui inclut une procédure pour rafraîchir la page web si un certain délai est passé après cette identification.

Procédure :

1) Après avoir développé dans votre IDE (PyCharm ou Thonny par exemple) le code demandé,
2) copiez ce code pour le coller dans la sous-fenêtre UpyLaB correspondante ;
3) cliquez sur le bouton « Valider ».
4) Si le code passe les tests, félicitations vous obtenez les points correspondants.
5) Si les tests ne sont pas concluants, vous devez corriger votre code et recommencer la procédure.
6) Si par contre un message d'erreur apparaît (Erreur serveur : session expirée ou non-trouvée. Veuillez rafraîchir la page dans le navigateur),
   - cliquez sur le bouton `Rafraîchir la page si demandé par UpyLaB avant de vérifier à nouveau`
   - et ensuite cliquez à nouveau sur le bouton `Valider`
   - et reprenez la procédure au point 4.

La fenêtre UpyLaB reste bloquée sur Connexion en cours . . .

Pour certains navigateurs Web et certaines configurations, des problèmes d'accès à notre outil UpyLaB peuvent survenir. Typiquement, la fenêtre reste bloquée avec un message `Connexion en cours ...`

Si cela se produit, cliquez simplement sur le bouton `Rafraîchir la page si demandé par UpyLaB avant de vérifier à nouveau` situé en dessous de l'exercice (même si ce n'est pas demandé explicitement :-)).

Si le souci perdure, une des causes les plus courantes est l'utilisation de code anti « pop-up » ou anti-mouchard (en particulier avec Firefox). Si c'est le cas, il est possible que vous deviez autoriser l'accès à https://upylab.ulb.ac.be (l'adresse d'accès à notre outil UpyLaB).

Donc si une telle erreur se produit, allez voir dans la FAQ (Foire Aux Questions) (onglet FAQ) ou cliquez sur l'onglet `Discussion` et trouvez le fil de discussion `Problèmes d'accès à UpyLaB` où un résumé des soucis constatés et des solutions précises proposées y a été déposé.

COMMENT RÉSOUDRE LES EXERCICES UPYLAB

Un point essentiel à fixer pour bien apprendre lors de ce cours est la façon de réaliser les exercices UpyLaB. Pour chaque exercice UpyLaB, l'énoncé est fourni.

UpyLaB n'est pas un environnement de développement. Il permet juste de tester si votre code lui semble correct.

Ainsi, après avoir bien compris l'énoncé de l'exercice, il est important de développer une solution en utilisant l'IDE PyCharm (ou si PyCharm n'est pas disponible, IDLE ou Trinket) ou, pour voir de façon détaillée comment votre code s'exécute, l'outil Python Tutor que nous présentons à la section suivante. Ce n'est que quand votre solution sera complète et que vous l'aurez validée en testant le code que vous pourrez copier et coller ce code dans la fenêtre UpyLaB de l'exercice pour lancer la validation. Tester son code signifie exécuter le programme plusieurs fois et, si possible, sur des exemples différents, pour être convaincu qu'il donne toujours une réponse correcte.

Si UpyLaB ne le valide pas - j'ai par exemple mis l'instruction :

```
print("Bonsoir UpyLaB !")
```

ou même

```
print("Bonjour UpyLaB ! ")
```

avec **une espace en trop après le point d'exclamation**,

à la place de :

```
print("Bonjour UpyLaB !")
```

vous devez le corriger avant de soumettre une solution modifiée.

Attention : Le nombre de vérifications que vous pouvez faire avec UpyLaB n'est pas limité. Malgré tout, il est préférable de réussir l'exercice avec un nombre de clics « Valider » le plus petit possible.

Nous insistons sur le fait qu'UpyLaB est un exerciseur avec un environnement de tests et non un environnement de développement. Prenez donc l'habitude de développer chacun de vos codes dans un environnement de développement comme PyCharm, et ensuite, quand vous le jugez correct et complet, de faire un copier-coller dans la fenêtre UpyLaB qui doit recevoir le code qui solutionne le problème, pour ensuite cliquer sur le bouton « Valider » afin de réaliser les tests.

1.4.3 L'outil Python Tutor

PYTHON TUTOR

Python Tutor est un outil en ligne créé et maintenu par Philip Guo de l'Université de San Diego. Comme tout ce que nous utilisons pour ce cours, son utilisation est totalement libre et gratuite.

Python Tutor permet d'exécuter pas à pas des petits scripts Python en visualisant l'effet de chaque instruction Python exécutée, comme le montre le petit exemple ci-dessous. Dans notre cours, nous utiliserons cet outil **de façon intégrée** à nos pages.

Exemple Python Tutor du code Python qui affiche les trois lignes suivantes avant de se terminer

```
Hello World
Bonjour le Monde !
J'apprends Python 3
```

Python Tutor va nous permettre de détailler, instruction par instruction, comment Python fonctionne en cliquant sur les boutons « Next » (et « Prev » pour revenir en arrière) ou en actionnant le curseur.

N'hésitez pas à animer l'exemple ci-dessous en cliquant sur ces boutons et curseur !

Exemple Python Tutor intégré

```
print('Hello World')
print('Bonjour le Monde !')
print("J'apprends Python 3")
```

Note : Voir animation du code avec Python Tutor dans le cours en ligne

Utilisation de Python Tutor

Si, durant ce cours ou plus tard, vous désirez bien comprendre un code Python, vous pouvez accéder à l'outil Python Tutor en ligne via la page http://pythontutor.com pour l'utiliser quand cela vous semble utile. Attendons d'avoir vu plus de concepts de Python, au module suivant, pour en faire une démonstration plus complète et bien comprendre son utilité pour l'apprentissage du langage.

Pour l'instant ces concepts sont probablement assez flous pour vous. Ils deviendront plus clairs dès le module suivant quand nous commencerons à expliquer les instructions de base.

1.4.4 La documentation officielle sur python.org

DOCUMENTATION OFFICIELLE

Dans le cadre de ce cours, vous n'aurez normalement pas besoin de consulter la documentation « officielle » Python. En effet la documentation officielle d'un langage de programmation est souvent un peu rédhibitoire pour des débutants. Malheureusement, parfois, c'est le seul endroit où l'on trouve ce que l'on cherche.

L'accès à cette documentation se fait via le site officiel de Python (https://python.org) qui contient un menu « documentation » : cliquez sur le bouton « Python 3.x Docs » qui vous propose toute la documentation officielle Python 3, y compris une possibilité en haut à droite de faire une recherche rapide. Essayez cette recherche, et vous verrez qu'au début vous serez probablement noyé dans l'abondance d'information. Ne vous en faites pas pour cela !

1.5 Au fait, c'est quoi un code ou un programme ?

1.5.1 Quelques définitions

INFORMATION, DONNÉE, ALGORITHME : QU'ES AQUÒ ?

Il est important pour tout un chacun de connaître quelques mots de « jargon » et quelques définitions dans le domaine. Cela permet de mieux comprendre et connaître les informaticiens et leur travail, mais aussi de comprendre leur façon de s'exprimer par exemple lorsqu'ils décrivent un problème à résoudre.

Commençons par brièvement parler de l'informatique. Le but de l'informatique est d'effectuer du **traitement** automatisé de l'**information**.

L'**information** est un ensemble d'éléments qui ont une signification dans le contexte étudié.

Les **données** d'un problème sont représentées par l'ensemble des informations utilisées pour résoudre ce problème en vue d'obtenir les **résultats** escomptés. Pour cela, l'informaticien peut commencer par écrire des algorithmes.

Un **algorithme** n'est pas conçu uniquement pour obtenir un résultat pour une donnée bien précise, mais constitue une **méthode** qui permet, à partir de n'importe quelle autre donnée du même type, d'obtenir le résultat correspondant.

Un exemple simple d'algorithme est celui qui consiste, depuis l'entrée, à trouver la sortie d'un labyrinthe.

— Les données de l'algorithme sont le plan du labyrinthe avec en particulier l'endroit où se trouvent l'entrée et la sortie.
— L'algorithme va consister à entrer dans le labyrinthe et ensuite longer le côté gauche (ou le droit mais sans alterner) et avancer tant que possible, en faisant demi-tour quand nous sommes bloqués mais en continuant à longer le côté gauche. Si le labyrinthe n'a qu'une seule entrée et une sortie sur les côtés extérieurs de celui-ci et qu'il n'a ni pont ni tunnel, cet algorithme permet de trouver la sortie.

— Le résultat de cet algorithme sera par exemple la séquence de mouvements à réaliser pour, depuis l'entrée, trouver la sortie.
Un algorithme peut être **implémenté** sur un ordinateur. Celui-ci ne possède jamais qu'une quantité limitée de mémoire de stockage d'information dont la précision est limitée. De ce fait pour résoudre certains problèmes qui, en théorie, pourraient requérir un calcul trop long, ou une précision ou un stockage d'information trop important, des algorithmes ne donnant qu'une valeur approchée du résultat doivent être conçus.

Dans le contexte de ce cours, on parle de code et de programme informatique sous forme de séquence d'instructions Python 3.

Un **code** est un programme informatique, une partie de programme, ou une liste d'instructions ou de commandes (un "script") pour le système de votre ordinateur.

Un programme est un ensemble d'instructions (donc du code) qui, quand il est exécuté sur un ordinateur, réalise un traitement défini. Un programme est donc vu comme la traduction en un code compréhensible par l'ordinateur d'un algorithme qui solutionne un problème.

Même si on peut faire de longs débats pour savoir si un bout de code peut être défini comme un programme, et plus important, pour savoir si quelqu'un peut être décrit comme un codeur ou un programmeur (faites par exemple une recherche sur le Web : "différence codeur programmeur"), dans notre contexte, nous supposons que c'est presque la même chose.

Un langage de programmation comme le langage Python définit les règles nécessaires pour le code ou le programme pour qu'il soit compréhensible et exécutable par un ordinateur.

Un programme appelé interpréteur ou compilateur « traduit » ce code source, c'est-à-dire dans le langage de programmation, en code machine. L'interpréteur « exécute » immédiatement chaque instruction analysée, tandis que le compilateur traduit d'abord complètement le code source en code machine qui pourra par la suite être exécuté.

Dans le contexte Python, vous pouvez voir une instruction comme un ordre que l'interpréteur Python donne à l'ordinateur qui exécute le code.

Écrire un code ou un programme ou a fortiori développer un gros logiciel demande une démarche en plusieurs étapes, appelée processus de développement d'un programme, qui peut être divisée en plusieurs phases (partiellement) successives.

— Analyse et spécification de ce qui est requis
— Conception
— Implémentation
— Tests et installation
— Exploitation et maintenance

Chaque phase produit des résultats écrits : spécification de ce qui est requis (cahier de charges), manuel utilisateur, description du fonctionnement, description succincte ou détaillée de l'algorithme, programme dûment commenté, historique des modifications, ...

Le travail d'un ingénieur système est de mener ces phases ou de les superviser.

Dans ce cours nous nous concentrons sur l'implémentation, c'est-à-dire le codage qui, bien sûr, demande de travailler sur les quatre premières phases de développement logiciel.

1.5.2 Quiz de fin de module

QUIZ DE FIN DE MODULE

Tous les quiz de la formation comptent pour la note finale du cours en ligne. Attention, pour certains quiz, comme celui-ci, vous aurez droit à plusieurs essais, mais pour la plupart, vous n'aurez droit qu'à un seul essai !

Note : Voir le quiz en section 1.5.2 du cours en ligne

POUR FINIR UN PETIT SONDAGE SUR L'INSTALLATION DE PYTHON

Note : Sondage sur l'installation de Python 3 et de PyCharm Community : voir en section 1.5.2 du cours en ligne

Retour d'information

- Si vos installations de Python 3 et PyCharm ont été **totalement** aisées : chouette on peut y aller ; merci de voir dans le forum si vous pouvez aider un apprenant qui aurait un souci.
- Si vos installations de Python 3 et PyCharm ont été **globalement** aisées : nous espérons que cela ne vous a pas pris trop de temps, on continue ; merci de voir dans le forum si vous pouvez aider un apprenant qui aurait le même type de souci que celui que vous avez expérimenté.
- Si votre installation de Python 3 et de PyCharm ont été **difficiles** : nous espérons que cela ne vous a pas pris trop de temps, on continue ; merci éventuellement de dire dans le forum comment votre souci a été résolu.
- Si **vous n'êtes pas arrivé** à installer Python 3 ou PyCharm : si personne de votre entourage ne peut vous aider, utilisez le forum ; nous espérons que quelqu'un pourra y solutionner vos soucis.

1.6 Références et bilan du module

1.6.1 Références et bilan

RÉFÉRENCES

Dans ce module nous avons planté le décor. Vous avez installé l'environnement Python 3 avec PyCharm sur votre ordinateur et pris connaissance d'outils qui vont nous aider tout au long du cours ; les références qui suivent complètent la panoplie d'outils.

Références :

- http://pythontutor.com : accès à "Start visualizing your code now" qui vous permet de visualiser comment s'exécute pas à pas votre code (ne pas oublier de mettre l'option python 3.6 ou ultérieur)
- Si vous désirez avoir un livre complet en plus du support de cours, le livre de Gérard Swinnen Apprendre à programmer avec Python 3 est une excellente référence pour débuter votre apprentissage
- https://www.python.org : site officiel de Python (téléchargements et documentations Python 3)

BILAN DU MODULE

Note : Voir la vidéo de la section 1.6.1 du cours en ligne

RÉSUMÉ DE LA VIDÉO

- Vous avez installé et fait connaissance avec Python 3 ainsi que l'environnement PyCharm.
- Vous avez vu comment soumettre un exercice UpyLaB et comment manipuler une démonstration Python Tutor.
- En route vers le module 2 et l'apprentissage de la programmation ! Ce module 2 va nous montrer les bases de Python qui peut manipuler des nombres et faire de l'arithmétique mais aussi faire du traitement de textes.

CHAPITRE 2

Python comme machine à calculer et à écrire

2.1 Tout un programme au menu

2.1.1 Présentation du menu de ce module

MENU DE CE MODULE : MON PREMIER PROGRAMME COMPLET

Note : Voir la vidéo de la section 2.1.1 : Mon premier programme Python

PRÉSENTATION DU MODULE 2

Nous voici arrivés au module 2 de ce cours.

Jusqu'à présent nous avons juste installé le décor.

À partir d'ici, nous allons entamer l'apprentissage de la programmation et du langage Python.

Dans ce module, nous apprendrons comment faire de l'arithmétique et manipuler des textes. Nous verrons aussi comment facilement faire des dessins géométriques.

Dans les modules suivants, nous parlerons des instructions Python (module 3), nous verrons comment structurer un programme en utilisant des fonctions (module 4), et apprendrons à utiliser des structures de données plus compliquées (modules 5 et 6).

Au terme du cours en ligne, vous aurez appris la programmation de base et serez capable d'écrire des programmes élaborés, comme le témoignera le projet que vous aurez réalisé.

Pour chacune des notions que nous voulons vous apprendre, le principe sera un apprentissage en trois phases :

1) Nous vous donnerons les bases théoriques pour vous lancer.
2) Nous les mettrons en pratique généralement à travers de petits exercices encadrés.
3) Vous pourrez ensuite mettre ces connaissances en pratique grâce à de nombreux exercices.

Et n'oubliez pas que pour être à l'aise en programmation, il faut pratiquer intensivement. L'adage « C'est en forgeant que l'on devient forgeron » s'applique parfaitement ici.

2.2 Python comme machine à calculer

2.2.1 Valeurs et expressions

COMMENÇONS PAR L'ARITHMÉTIQUE

Note : Voir la vidéo de la section 2.2.1 : Valeurs et expressions

CONTENU DE LA VIDÉO

La vidéo précédente présente les opérations arithmétiques de base possibles en Python sur des valeurs entières (`int`) ou fractionnaires (`float`). Nous donnons ici les tests qui y sont réalisés sur la console Python suivis d'un résumé des explications données s'y rapportant.

Tests réalisés sur la console dans la vidéo

```
>>> 3
3
>>> 3 + 5
8
>>> 9 - 5
4
>>> 3 * 4
12
>>> 3 + 4 * 5
23
>>> (3 + 4) * 5
35
>>> 8 - 5 - 2
1
>>> 8 / 4
2.0
>>> 8 / 3
2.6666666666666665
>>> type(3)
<class 'int'>
>>> type(3 + 4 * 5)
<class 'int'>
>>> type(2.66)
<class 'float'>
>>> type(4.0)
<class 'float'>
>>> type(8 / 2)
<class 'float'>
>>> 8 // 4
2
>>> 8 // 3
2
>>> 10 ** 5
100000
>>> 3.14159
3.14159
>>> 314159e-5
3.14159
>>> 0.00001
```

(suite sur la page suivante)

(suite de la page précédente)

```
1e-05
>>> 1.0e-5
1e-05
>>> 2 ** 2 ** 3
256
```

La vidéo en bref

Si nous ouvrons une console Python (dans l'environnement PyCharm), celle-ci attend une instruction de l'utilisateur. On voit les symboles >>> qui invitent l'utilisateur à entrer quelque chose.

Dans ce mode, appelé mode interactif, la valeur de ce que l'on évalue est renvoyée. On peut ainsi voir l'effet des différentes opérations arithmétiques et la façon de les écrire : (addition `+`, soustraction `-`, multiplication `*`, division en nombre flottant `/`, division entière `//`, exponentiation `**`), ainsi que les types entiers (`int` : `3`, `8`, `12`, ...) et fractionnaires appelés aussi flottants (`float` : `2.0`, `2.666666666666665`, `3.14159`, `0.00001`, `1.0e-5`, ...).

2.2.2 L'arithmétique

TESTER L'ARITHMÉTIQUE

À vous maintenant de « tester les choses » : « tester » dans le jargon informatique signifie exécuter des petits programmes ou des instructions pour voir comment les choses se déroulent (donc ici comment l'interpréteur fonctionne).

Dans la console de Thonny, ou de PyCharm (menu Tools -> Python Console) et expérimentez ce que donnent des calculs utilisant les opérateurs arithmétiques sur les valeurs de type entier (`int`) et fractionnaire (`float`).

Si l'on regarde dans la documentation Python (The Python Standard Library sur le site python.org) la liste des opérateurs arithmétiques de base, on obtient (principalement)

`+`	L'addition
`-`	La soustraction
`*`	La multiplication
`/`	La division réelle (c'est-à-dire dont le résultat est du type float)
`//`	La division entière tronquée (une explication sur la division est donnée ici)
`**`	L'exponentiation (appelée également puissance ; une explication sur l'exponentiation est donnée ici)
`%`	Le modulo (appelé aussi modulus ; pour les nombres entiers positifs, le modulo est défini comme le reste de la division entière ; une explication du modulo est donnée ici).

Nous n'avons pas encore parlé de l'opérateur modulo écrit `%` en Python. Ici nous vous demandons d'être curieux pour comprendre comment fonctionne cet opérateur. La curiosité est un outil essentiel pour bien apprendre à programmer !

De plus Python en mode interactif dans une console vous facilite souvent la vie.

Par exemple, si vous saisissez `8 % 3` dans une console, vous obtiendrez bien 2, qui correspond au reste de la division entière de 8 par 3.

L'opérateur modulo Python fonctionne également avec des nombres négatifs et même avec des nombres fractionnaires.

ASSOCIATIVITÉ ET PRIORITÉ DES OPÉRATEURS

L'associativité et la priorité (appelée également précédence) de ces opérateurs (voir ci-dessous pour une explication), depuis le plus prioritaire vers le moins prioritaire, sont les suivantes :

`(expression)`	
`**`	associatifs à droite
`* / //`	associatifs à gauche
`+ -`	associatifs à gauche

où les opérateurs sur une même ligne ont le même niveau de priorité.

Priorité

Par exemple, la multiplication, appelée également produit, est plus prioritaire que l'addition : ainsi l'évaluation de `3 + 4 * 5` vaut `23` (on effectue d'abord la multiplication `4 * 5` et ensuite l'addition).

Notons que pour `2 ** -1` le moins est unaire (en fait il faut le voir comme le nombre `-1`) et donc vaut `2 ** (-1)` c'est-à-dire `0.5`.

Associativité

La plupart des opérateurs sont associatifs à gauche. La soustraction `-` est associative à gauche : ainsi, `8 - 5 - 2` est équivalent à `(8 - 5) - 2`.

Par contre, l'exponentiation `**` est associative à droite : ainsi `2 ** 2 ** 3` est équivalent à `2 ** (2 ** 3)`, c'est-à-dire `256` et non à `(2 ** 2) ** 3` qui vaut `64`.

Parenthèses

Des parenthèses peuvent être utilisées pour modifier l'ordre d'évaluation dans une expression ; ainsi : `(3 + 4) * 5` pour réaliser l'addition `(3 + 4)` avant la multiplication par `5`. On évalue donc prioritairement ce qui à l'intérieur des parenthèses.

QUIZ SUR L'ARITHMETIQUE

Dans le quiz suivant, nous vous demandons de nous dire, éventuellement avec l'aide d'une console Python, les résultats de différentes expressions utilisant l'opérateur modulo ou les autres opérateurs arithmétiques.

Note : Voir le quiz de la section 2.2.2

2.2.3 Tester l'arithmétique

PRÉPAREZ UNE MOUSSE AU CHOCOLAT !

Nous avons vu comment faire des calculs avec les différents opérateurs arithmétiques que Python nous fournit.

Avec ces opérateurs en poche, lançons-nous pour faire des calculs utiles !

Par exemple, faisons un peu de cuisine en réalisant la recette de la mousse au chocolat sur le site marmiton.org qui est donnée ici :

Ingrédients (pour 4 personnes)
- 3 oeufs
- 100 g chocolat (noir ou au lait)
- 1 sachet de sucre vanillé

Préparation de la recette
- Séparer les blancs des jaunes d'oeufs
- Faire ramollir le chocolat dans une casserole au bain-marie
- Hors du feu, incorporer les jaunes et le sucre

— Battre les blancs en neige ferme et les ajouter délicatement au mélange à l'aide d'une spatule
— Verser dans une terrine ou des verrines et mettre au frais 1 heure ou 2 minimum

QUIZ SUR LA MOUSSE AU CHOCOLAT

La page mousse au chocolat du site Marmiton donne la quantité de chaque ingrédient quand on adapte le nombre de personnes qui vont manger la recette.

On voit par exemple que pour 7 personnes, 6 oeufs, 175 g de chocolat et 1.75 sachet de sucre vanillé sont requis. Cela correspond aux quantités d'ingrédients pour 4 personnes, divisées par 4 pour calculer les quantités requises par personne, et multipliées par 7 correspondant aux 7 personnes. Si ce calcul pour les oeufs ne tombe pas juste (c'est-à-dire s'il existe une partie fractionnaire), la recette demande un oeuf de plus (c'est ce que l'on appelle la valeur plafond).

Mais si, pour simplifier, nous désirions calculer les ingrédients à l'unité près mais en faisant une simple troncature, c'est-à-dire en ne gardant que la valeur entière, quels résultats aurions-nous ?

Par exemple, les troncatures de 1.3 et de 1.6 donnent tous les deux la valeur 1.

Note : Pour obtenir la valeur tronquée à l'unité, on pourra utiliser l'opérateur de division entière //, en faisant attention aux priorités opératoires. Mais vous pouvez aussi utiliser la fonction prédéfinie `int()`. Par exemple int(1.6), qui reçoit comme argument la valeur 1.6 de type fractionnaire (float), donnera la valeur entière 1.

Donc, en utilisant une console Python, pouvez-pour me dire la quantité de chaque ingrédient que je dois avoir pour faire ma recette pour 7 personnes, en tronquant les valeurs calculées à l'unité près ?

Note : Les bons cuisiniers savent que Marmiton a raison dans ses calculs de proportions pour le nombre d'oeufs requis :-)

QUIZ

Note : Voir le quiz de la section 2.2.3

2.3 Python comme machine de traitement de texte

2.3.1 Les expressions chaînes de caractères

LES EXPRESSIONS CHAÎNES DE CARACTÈRES

La vidéo qui suit présente les notions de base sur les textes appelés chaînes de caractères Python.

VIDÉO SUR LES EXPRESSIONS CHAÎNES DE CARACTÈRES

Note : Voir la vidéo de la section 2.3.1 : Valeurs et expressions textuelles

CONTENU DE LA VIDÉO

La vidéo précédente présente les chaînes de caractères Python. Nous donnons ici les tests qui y sont réalisés sur la console Python suivis d'un résumé des explications données s'y rapportant.

Tests réalisés sur la console dans la vidéo

```
>>> "Bonjour"
'Bonjour'
>>> 'Bonjour'
'Bonjour'
>>> 'c'est facile'
  File "<input>", line 1
    'c'est facile'
        ^
SyntaxError: invalid syntax
>>> "c'est facile"
"c'est facile"
>>> "Bonjour " + "Michelle"
'Bonjour Michelle'
>>> "bon" * 10
'bonbonbonbonbonbonbonbonbonbon'
>>> 10 * "ha"
'hahahahahahahahahaha'
>>> len("Bonjour")
7
```

La vidéo en bref

`"Bonjour"` et `'Bonjour'` entourés par des doubles ou simples apostrophes, appelés aussi doubles ou simples quotes, sont des chaînes de caractères (textes).

`type('Bonjour')` exprime que le type (la classe) de la valeur `'Bonjour'` est `str` pour *string*, terme anglais pour chaîne de caractères, ou plus simplement texte.

Taper dans la console Python `'c'est facile'` renvoie un message d'erreur du type `SyntaxError` puisque l'interpréteur considère que la chaîne de caractères est `'c'` et ne comprend plus la suite.

On peut écrire `"c'est facile"` entourés de double quotes pour obtenir une chaîne de caractères correcte.

Concaténation et répétition

`"Bonjour " + "Michelle"` renvoie `"Bonjour Michelle"`

La concaténation (+) est l'opération qui consiste à coller deux textes ensemble pour n'en former qu'un.

`"bon" * 10` donne `'bonbonbonbonbonbonbonbonbonbon'` et `10 * "ha"` donne `'hahahahahahahahahaha'`

C'est la concaténation de 10 fois le texte `"bon"` ou de 10 fois le texte `"ha"`.

len()

La fonction prédéfinie `len()` donne la longueur de la séquence donnée en argument. Cela signifie que `len("Bonjour")` donne la longueur du texte, c'est-à-dire ici 7 caractères (les doubles quotes étant là pour marquer le début et la fin du texte).

2.3.2 À vous de tester les chaînes de caractères

TESTER LES CHAÎNES DE CARACTÈRES AVEC UNE CONSOLE PYCHARM

À vous de jouer ! Entrez à nouveau dans une console Python et expérimentez comment l'interpréteur réalise les manipulations simples de chaînes de caractères.

Pour ce faire, nous vous proposons de regarder ce que donnent les lignes de code suivantes :

```
>>> "bonjour"
>>> 'bonjour'
>>> '"  Bonjour "'
>>> "C'est facile"
```

Il est également possible d'appliquer des fonctions à des chaînes de caractères. Par exemple, la fonction prédéfinie `len()` donne la longueur de la séquence passée en argument. Cela signifie que `len("bonjour")` donne la longueur du texte soit 7 caractères (les doubles quotes étant là pour marquer le début et la fin du texte).

Afin de voir si vous avez compris l'utilité de la fonction `len()`, nous vous proposons le quiz suivant.

QUIZ SUR LA FONCTION LEN

Note : Voir le quiz de la section 2.3.2

2.4 Les variables pour changer

2.4.1 Les variables

LES VARIABLES POUR RETENIR LES TRAITEMENTS

La vidéo qui suit présente les notions fondamentales de variables et *d'assignation*, appelée également *affectation*, en Python.

VIDÉO SUR LES VARIABLES

Note : Voir la vidéo de la section 2.4.1 : Variable et assignation

CONTENU DE LA VIDÉO

La vidéo précédente présente les variables Python. Nous donnons ici les tests qui y sont réalisés sur la console Python suivis d'un résumé des explications données s'y rapportant.

Tests réalisés sur la console dans la vidéo

```
>>> x = 3
>>> x
3
>>> y = 4 * 3
>>> y
12
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> y = 4
>>> y
4
>>> x + 5
8
>>> z = x + 5
>>> y = y + 1
>>> y
5
>>> r = 8 / 3
>>> r
2.6666666666666665
>>> type(y)
<class 'int'>
>>> type(r)
<class 'float'>
>>> mon_message = 'bonjour'
>>> x, y, z
(3, 5, 8)
>>> ma_variable = x
>>> ma_variable
3
```

La vidéo en bref

Les variables permettent de retenir des valeurs ; l'instruction d'*affectation*, appelée également *assignation*, donne une valeur à une variable.

Une assignation a la forme suivante

`nom = valeur`

où **à gauche** du symbole d'assignation =, `nom` est le nom de la variable, choisi par le programmeur, et **à droite** du symbole = est donnée la `valeur` à assigner. Par exemple :

```
x = 3
y = 4 * 3
z = x + 5
y = y + 1
r = 8 / 3
mon_message = 'bonjour'
```

Après l'assignation, (par exemple `x = 3`), la variable assignée a une valeur et un type (ici `x` *vaut* la valeur *entière* 3) ;

`x + 5` utilise la valeur de la variable `x` pour faire les calculs (ici comme `x` vaut `3` : `3 + 5` c'est-à-dire `8`) ;

`type(x)` renseigne que la variable `x`, donnée en argument de la fonction prédéfinie `type()` est de type entier (son contenu est un entier). De façon raccourcie on dit que `x` est un entier.

Certains noms sont des mots réservés (mots-clés) Python : ainsi `if = 0` ne fonctionne pas : l'interpréteur donne une erreur de syntaxe car `if` est un mot-clé Python.

Notons plus précisément qu'une valeur en Python sera un *objet* et une variable le *nom* d'un objet.

Avertissement : La variable qui reçoit la valeur d'une assignation est à gauche du symbole = (symbole d'affectation).

N'essayez pas d'écrire

```
3 = x
```

car l'interpréteur vous donnera une erreur de syntaxe

```
SyntaxError: cannot assign to literal
```

qui exprime que vous essayez d'assigner la valeur de la variable `x` à la valeur 3, ce qui n'est pas possible !

2.4.2 Code avec variables

À VOUS DE JOUER !

L'exercice suivant vous demande d'écrire votre premier code utilisant des variables.

Pour cela, nous allons reprendre la préparation de la mousse au chocolat donnée dans l'activité 2.2.3.

Nous vous demandons de calculer à nouveau les quantités nécessaires de chaque ingrédient mais cette fois, pour un nombre quelconque de personnes.

Votre programme commencera donc par assigner le nombre de convives à une variable (de nom) `n`. Ensuite, il calculera les quantités (en grammes ou en unités) de chaque ingrédient qu'il stockera dans trois autres variables : `oeufs`, `chocolat` et `sucre_vanille`. Chacune de ces quantités sera, comme dans l'activité précédente, tronquée à l'unité près. Par souci de simplification, on supposera que le nombre de personnes pour lesquelles il faut préparer la mousse au chocolat sera toujours supérieur ou égal à 4.

Pour `n` valant 7, les valeurs pour `oeufs`, `chocolat` (en grammes) et `sucre_vanille` sont respectivement de 5, 175 et 1.

On voit que l'assignation en Python revient réellement à donner un nom à une valeur.

PROPOSITION DE SOLUTION

Vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir une autre solution : nous vous en proposons une.

```
n = 12
oeufs = 3 * n // 4
chocolat = 100 * n // 4
sucre_vanille = n // 4
```

RÈGLE DE BONNE PRATIQUE

À toute fin utile, voici une règle de bonne pratique qui vous permettra de rendre votre code plus lisible et souvent plus efficace : on utilise généralement des noms de variables qui ont un sens en fonction de leur contenu. Par exemple sucre_vanille. En Python, les noms des variables ne peuvent contenir d'accents ni de caractères non alphanumériques (un caractère alphanumérique signifie une lettre ou un chiffre) excepté le caractère souligné « _ », et ne peuvent pas commencer par un chiffre. Par exemple `sucre_vanille`, `s9` ou `oeufs` sont des noms corrects pour des variables mais par exemple `sucre_vanillé` (qui contient un accent) ou `9s` (qui commence par un chiffre) ne sont pas des noms de variables valides.

Note : PyCharm analyse nos scripts pour déterminer s'ils respectent bien les « règles de bonnes pratiques » (nous en reparlerons plus loin) : si les mots utilisés par exemple pour donner un nom aux variables ne sont pas dans ses dictionnaires, PyCharm nous avertit qu'il y a probablement un typo (mais peut-être que vous l'avez délibérément orthographié ainsi). C'est pour cette raison que nous avons ajouté un dictionnaire français contenant les mots avec et sans accents ce qui permet de donner dans nos codes des noms de variables sans les accents (par exemple `cafe = 5` si notre programme parle de café).

UN PEU DE VOCABULAIRE : VARIABLE ET CONSTANTE

Dans un programme, on distingue les valeurs qui peuvent changer, comme la valeur des variables `n`, `oeufs`..., des valeurs qui restent inchangées tout au long du programme, et que l'on appelle *constantes* dans le jargon informatique. Comme son nom l'indique la valeur d'une constante n'est jamais modifiée lors de l'exécution du programme. Ainsi, de façon évidente, les valeurs 0, 1 ou 7 sont des constantes. Plus loin, on parlera de la valeur pi comme étant une constante.

SYNTAXE ET SÉMANTIQUE

Rappelons la définition de ces deux mots courants en informatique et dans notre cours en ligne.

Syntaxe : pour un programme Python, est l'ensemble des règles grammaticales que l'on doit suivre pour que le programme soit a priori compréhensible par l'interpréteur. Parmi les règles de syntaxe à suivre, nous avons vu que l'assignation doit avoir un nom de variable à gauche du signe « = » et une valeur ou une expression à droite.

Toute erreur syntaxique dans l'écriture de code Python sera sanctionnée par l'interpréteur, lors de l'exécution de la partie incriminée par un message : **SyntaxError** avec une explication du soucis.

Sémantique : exprime le « sens », la « signification ». Parmi les erreurs sémantiques dans un code Python, on a par exemple le fait d'utiliser un nom (identificateur) non défini.

Par exemple `x = z` est une instruction Python syntaxiquement correcte. Mais, si z n'a encore reçu aucune valeur, par exemple, via l'assignation (z n'a pas encore été définie), l'interpréteur Python qui exécute cette instruction produira une erreur (`NameError`) : c'est une erreur sémantique.

Note : Voir le quiz de la section 2.4.3

2.5 PyCharm en mode script, entrées et sorties

2.5.1 Manipuler des scripts avec PyCharm

DES SCRIPTS QUI COMMUNIQUENT : C'EST MIEUX !

La capsule suivante introduit les deux fonctions `input` et `print` qui seront bien utiles pour que le code Python puisse communiquer avec son utilisateur. La notion de script Python est aussi introduite.

VIDÉO SUR LES SCRIPTS

Note : Voir la vidéo de la section 2.5.1 : print et input

CONTENU DE LA VIDÉO

La vidéo précédente présente les scripts Python ainsi que les foncctions `input` et `print`. Son résumé est groupé avec celui de la vidéo de l'onglet 3 de cette section, un peu plus loin dans ce cours ; cette seconde vidéo complète l'explication. Le résumé des deux vidéos est disponible dans cet onglet 3.

2.5.2 Python Tutor et les diagrammes d'état

UTILISATION DE PYTHON TUTOR

Nous avons déjà présenté Python Tutor dans le module précédent. Python Tutor est un outil en ligne permettant d'exécuter pas à pas des petits scripts Python en visualisant l'effet de chaque instruction Python.

Avant de montrer de façon plus précise comment fonctionne Python Tutor, commençons par expliquer pourquoi diable nous introduisons un outil de plus dans ce cours en ligne. La réponse est double :

— d'une part Python Tutor permet d'interpréter le code Python en s'arrêtant à chaque instruction pour voir son effet,
— d'autre part, après chaque instruction, Python Tutor représente le « diagramme d'état » illustrant l'état du programme à ce moment-là.

Comme nous allons le voir, un diagramme d'état est un schéma qui montre graphiquement comment les variables et valeurs manipulées dans un script Python sont stockées en mémoire. Si au départ cette information ne vous sera pas d'une grande utilité, petit à petit nous verrons qu'elle est essentielle pour comprendre comment nos codes fonctionnent.

Python Tutor n'est donc pas un simple interpréteur ; c'est un outil d'apprentissage de Python (et d'autres langages) principalement pour les débutants. Python Tutor nous sera d'une grande utilité, en particulier pour visualiser les diagrammes d'état montrant l'état de codes fournis, à chaque étape de l'exécution, ce qu'un interpréteur « normal » comme PyCharm ne fait pas. Mais Python Tutor n'est pas un interpréteur complet de scripts en ligne comme ce qui vous est fourni via PyCharm. Ainsi, Python Tutor ne permet l'exécution que des scripts utilisant un petit sous-ensemble d'instructions Python.

Pour bien expliquer les nouveaux concepts Python, nous inclurons fréquemment des exemples Python Tutor intégrés dans ce cours en ligne. Mais libre à vous de mettre dans une fenêtre Python Tutor un code Python de votre choix pour visualiser son exécution étape par étape et bien comprendre son fonctionnement grâce aux diagrammes d'état fournis par Python Tutor.

Pour vous aider à utiliser Python Tutor, la petite vidéo qui suit fait une démonstration de son utilisation. Libre à vous de la visionner maintenant ou quand vous en aurez besoin.

RÉSUMÉ DE LA MANIÈRE D'UTILISER PYTHON TUTOR

— Sur votre navigateur Web ouvrez la page pythontutor.com.
— Cliquez sur "Start visualizing your code now" qui vous permet de visualiser comment s'exécute, pas à pas, le script que vous lui donnez dans la zone centrale de la fenêtre.
— N'oubliez pas de mettre l'option python 3.6 ou ultérieur et de sélectionner les bonnes options proposées sous cette fenêtre. Nous vous conseillons de prendre les options par défaut, et si vous désirez que les diagrammes d'état fournissent plus de détails sur les valeurs, remplacez l'option « inline primitives & nested objects » par « render all objects on the heap ».
— Vous pouvez ensuite exécuter pas à pas en avant ou en arrière le script en cliquant sur le bouton « Visualize Execution » et ensuite utiliser les boutons « Next », « Back », « First », « Last » ou encore le curseur.
— L'affichage résultant des print est donné dans une sous-fenêtre « Print output », et lors des input une fenêtre sous l'intitulé « Enter user input below : » s'ouvre vous permettant d'entrer la donnée après avoir cliqué sur le bouton « submit ».
— Pour revenir en mode édition, cliquez sur le bouton « Edit the code ».

Dans les exemples Python Tutor intégrés au code, toute la partie initialisation a déjà été réalisée pour vous. Il ne vous reste plus qu'à cliquer sur les boutons pour faire avancer ou reculer l'exécution ainsi qu'à introduire les inputs demandés.

La vidéo qui suit illustre ce qui vient d'être expliqué.

PYTHON TUTOR

Note : Voir la vidéo de la section 2.5.2 : Python Tutor

À VOUS DE JOUER !

Dans le but d'apprendre à manipuler Python Tutor, reprenez votre script rédigé à la section précédente qui calcule les quantités d'ingrédients pour la préparation de la mousse au chocolat. Modifiez-le dans Python Tutor pour lire la valeur de `n` (`input` de n) et écrire les résultats grâce à des `print` : la valeur des variables `oeufs`, `chocolat` et `sucre_vanille` .

Ensuite, visualisez comment celui-ci s'exécute pas à pas dans une fenêtre de Python Tutor.

ANIMATIONS PYTHON TUTOR INTÉGRÉES

Pour rappel, dans ce cours, nous utiliserons également Python Tutor de façon intégrée, c'est-à-dire au sein même des pages du cours. Par exemple, voici deux animations Python Tutor intégrées de l'exemple précédent sur la mousse au chocolat avec `input` et `print`. La première utilise l'option « render all objects on the heap », la seconde l'option « inline primitives & nested objects ».

Note : Voir exemples Python Tutor dans le cours en ligne

2.5.3 Commentons notre programme

DES SCRIPTS COMMENTÉS : C'EST ENCORE MIEUX !

Note : Voir la vidéo de la section 2.5.3 : Commentaires et docstrings

CONTENU DES VIDÉOS

Les deux vidéos sur les scripts Python ainsi que les fonctions `input` et `print`, présentées dans l'onglet précédent et l'onglet courant de la présente section, sont résumées ici.

Nous donnons ici les scripts qui y sont utilisés pour la présentation suivis d'un résumé des explications données s'y rapportant.

Scripts utilisés dans les vidéos

```
rayon = 5.0
circ = 2 * 3.14  rayon # l'astérisque manque
aire = 3.14 * rayon **  2
circ, aire
```

```
rayon = 5.0
circ = 2 * 3.14 * rayon
aire = 3.14 * rayon **  2
circ, aire
```

```
rayon = 5.0
circ = 2 * 3.14 * rayon
aire = 3.14 * rayon **  2
```

(suite sur la page suivante)

(suite de la page précédente)

```
circ, aire
print(circ, aire)
```

```
rayon = float(input())
circ = 2 * 3.14  * rayon
aire = 3.14 * rayon **  2
print(circ, aire)
```

```
rayon = float(input("Veuillez donner le rayon : "))
circ = 2 * 3.14  * rayon
aire = 3.14 * rayon **  2
print("Circonférence :", circ)
print("Aire          :", aire)
```

```
""" Auteur: Sébastien Hoarau
    date : juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque
             l'aire du disque
"""

rayon = float(input("Veuillez donner le rayon : "))  # lecture du rayon de mon disque
circ = 2 * 3.14  * rayon                    # calcul de la circonférence
aire = 3.14 * rayon **  2                   # calcul de l'aire
# Affichage des résultats
print("Circonférence : ", circ)
print("Aire          : ", aire)
```

La vidéo en bref

Les deux fonctions `input` et `print` sont introduites. Elles sont bien utiles pour que le code Python puisse communiquer avec son utilisateur. La notion de script Python est aussi introduite.

1) Le premier script est édité et ensuite sauvé par exemple dans le fichier `disque.py`. Son exécution par l'interpréteur Python donne une erreur de syntaxe : l'interpréteur lorsqu'il désire calculer la valeur de `circ` ne comprend pas ce qui est demandé. En effet cette ligne de code n'est pas correcte : il manque l'opérateur de multiplication `*`.
2) Après avoir corrigé mon script en ajoutant le symbole de multiplication, je redemande, avec la commande **Run**, à l'interpréteur d'exécuter le code du script `disque.py` et cette fois il se termine avec un `exit code 0` qui signifie que l'exécution s'est bien déroulée. Mais nous constatons que le script n'a pas communiqué de résultats !
3) Le troisième script permet avec la fonction `print` d'"*afficher*, on dit aussi *imprimer*, les résultats donnés en arguments du `print`.
4) Le quatrième script permet de recevoir grâce à la fonction `input()` une *donnée* que l'utilisateur peut encoder. La donnée reçue grâce à la fonction `input` est de type chaîne de caractères. Cette donnée est traduite dans ce script, en valeur fractionnaire grâce à la fonction prédéfinie `float()`. Notez l'écriture pour réaliser la succession lecture de la donnée qui donne une chaîne de caractères et traduction en valeur fractionnaire : `float(input())`.
5) Le cinquième script ci-dessus demande l'entrée de données (`input()`) avec un texte explicatif en argument, l'affichage de résultats (`print()`) également avec un texte explicatif à chaque print.
6) Le dernier script contient des *commentaires*. De façon générale, pour rendre un script plus lisible pour les autres programmeurs qui voudraient comprendre son fonctionnement, on peut rajouter des explications. Pour cela on peut, à la fin de chaque ligne de code, mettre le caractère # (croisillon) suivi de texte explicatif. Ce texte s'appelle du *commentaire* et ne

sera pas utilisé par l'interpréteur. Je peux aussi, n'importe où, rajouter un commentaire multiligne entouré de 3 simples quotes ou double quotes. Je peux mettre une explication plus complète de ce que fait le script dans un *docstring* initial :
— l'auteur
— la date
— le but du programme
— les données reçues (input)
— les résultats affichés (print)

les caractères croisillon et dièse

Le caractère # utilisé en Python pour marquer les commentaires est le caractère *croisillon* ; celui-ci a deux barres obliques parallèles coupées par deux barres horizontales. Ce caractère est souvent erronément appelé dièse (♯). Ce dernier, utilisé par exemple dans les partitions musicales, a deux barres verticales coupées par deux barres obliques parallèles.

CODE COMPLET COLORISÉ POUR L'EXEMPLE « MOUSSE AU CHOCOLAT »

Nous pouvons maintenant visualiser un script complet commenté. Pour illustrer cela, donnons deux scripts complets solutionnant le problème de la mousse au chocolat :
— une version avec assignation des résultats dans des variables
— et une version améliorée où les calculs se font directement lors des print.

Solution avec variables

```
"""Auteur: Thierry Massart
   Date : 5 décembre 2017
   But du programme : calcule les ingrédients nécessaires
   pour préparer de la mousse au chocolat pour n personnes
   Entrée: n (nombre de personnes)
   Sorties: nombre d'oeufs,
            quantité en gramme de chocolat,
            nombre de sachets de sucre vanillé
"""
n = int(input("nombre de personnes : "))        # entrée: nombre de personnes
oeufs = int(3*n/4)                              # nombre d'oeufs nécessaires
chocolat = int(100*n/4)                         # quantité de chocolat nécessaire
sucre_vanille = max(int(n/4), 1)                # quantité de sucre nécessaire
print("nombre d'oeufs : ", oeufs)               # affiche les résultats
print("quantité de chocolat (g) : ", chocolat)
print("quantité de sucre_vanillé : ", sucre_vanille)
```

Solution améliorée

```
"""Auteur: Thierry Massart
   Date : 5 décembre 2017
   But du programme : calcule les ingrédients nécessaires
   pour préparer de la mousse au chocolat pour n personnes
   Entrée: n (nombre de personnes)
   Sorties: nombre d'oeufs,
            quantité en gramme de chocolat,
            nombre de sachets de sucre vanillé
"""
n = int(input("nombre de personnes : ")) # entrée: nombre de personnes
print("nombre d'oeufs : ", int(3*n/4)) # calcule et affiche les résultats
print("quantité de chocolat en grammes : ", int(100*n/4))
print("quantité de sucre_vanillé : ", max(int(n/4), 1))
```

2.5.4 Réalisation des exercices UpyLaB du module

MISE EN PRATIQUE AVEC UPYLAB

Nous voici à la dernière étape d'apprentissage pour la matière de ce module : l'apprentissage autonome. Pour cela, nous vous demandons de réaliser les exercices 2.1 à 2.3 du Module 2 qui vous sont proposés dans le cadre de cette activité.

Note : En cas de problème d'affichage, essayez de recharger la page. Il se peut que votre navigateur ne sache pas bien afficher le code UpyLaB, et que vous deviez en utiliser un autre.

DÉVELOPPEMENT, MISE AU POINT ET DÉBOGAGE D'UN CODE

Faire les exercices UpyLaB va vous demander de *développer du code Python* résolvant les problèmes demandés. En général, au début, votre code ne résoudra pas (exactement) le problème demandé : nous dirons simplement que votre code est faux ! Dans ce cas, vous devrez le *déboguer*.

Déboguer son code (debug en anglais) correspond à y faire la chasse aux erreurs. La pratique de la programmation apprend comment déboguer efficacement un programme. Nous essayerons tout au long de ce cours de vous donner certains conseils pour y arriver. En voici quelques uns.

CONSEILS ET CONSIGNES LORS DE LA RÉALISATION D'EXERCICES UPYLAB

1) Lors du développement de chacun de vos codes, il est plus que vivement conseillé d'utiliser un environnement de développement comme **PyCharm** ou **Python Tutor** pour *tester* ou même *déboguer* pas par pas votre code avant de passer aux tests *UpyLaB*. En effet, faire tester directement par UpyLaB un code dont vous n'avez pas une certaine confiance dans le fait qu'il soit correct s'avère souvent être un gouffre au niveau du temps. En effet, pour valider votre code, UpyLaB ajoute du code supplémentaire qu'il va ensuite exécuter : si votre code est erroné à une ligne donnée, il est possible qu'UpyLaB évoque un autre numéro de ligne pour cette erreur ou renvoie un message peu explicite. Pour cette raison, il est recommandé de ne pas utiliser UpyLaB lors des développements de vos codes, mais uniquement pour valider s'ils sont bien corrects.
   La démarche la plus efficace pour développer du code en Python est donc :
   — de comprendre exactement ce qui vous est demandé,
   — de développer une solution dans PyCharm en la testant,
   — de tester l'ensemble de votre code dans UpyLaB (en le transférant dans UpyLaB grâce au copier / coller du code depuis PyCharm) pour faire valider votre script.
2) UpyLaB est parfois « psychorigide » dans le sens où, si votre code ne fait pas **exactement** ce qu'il demande, UpyLaB le considère comme faux. En particulier dans les consignes, votre code à tester par UpyLaB :
   — ne doit pas avoir d'arguments dans les appels à `input`.
   — ne doit pas imprimer autre chose que le résultat dans les `print`.
   Ainsi, il **faut** écrire :

```
a = float(input())
print(a)
```

   plutôt que :

```
a = float(input("a = "))
print('a vaut :', a)
```

   En effet UpyLaB teste votre programme en l'exécutant plusieurs fois. Durant ces tests, la sortie (`print`) du code proposé sur différentes entrées fournies automatiquement est comparée à la sortie attendue, produite par une solution de référence. Si votre code fait des sorties supplémentaires avec les fonctions `input` ou `print`, il sera jugé faux par UpyLaB lors des validations ! Vous expérimenterez sûrement cela tôt ou tard. Cela vous obligera à une rigueur à laquelle les novices en programmation sont généralement peu habitués.

Attention : Le but des exercices UpyLaB est de vous rendre **autonome** en programmation. C'est vous qui devez trouver la ou une solution, en code Python, à chaque exercice demandé. Si vous ne voyez pas comment y arriver ou qu'UpyLaB ne valide toujours pas votre code, voici quelques conseils :

1. Relire l'énoncé et les consignes associés à l'exercice ainsi que les éventuels conseils.
2. Lire la FAQ générale à propos des exercices UpyLaB.
3. Lire, si elle existe, la FAQ spécifique à l'exercice réalisé.
4. Si rien de tout cela ne fonctionne, poser votre question sur le Forum dédié aux exercices, en étant le plus précis possible, *sans mettre votre code complet ni substantiel* mais en mettant des copies d'écran des résultats fournis par UpyLaB.

2.5.5 Exercice UpyLaB 2.1 : Vert

Le but de cet exercice est de vérifier que vous savez définir des variables, et leur affecter des valeurs des différents types rencontrés dans le cours.

Écrire un programme qui assigne :
— la valeur entière 36 à la variable `x` ;
— la valeur entière résultat de 36 fois 36 à la variable `produit` ;
— la valeur entière résultat de la division entière de 36 par 5 à la variable `div_entiere` ;
— la valeur entière résultat de 15 exposant 15 à la variable `expo` ;
— la valeur float 3.14159 à la variable `pi` ;
— la valeur chaîne de caractères `"Bonjour"` à la variable `mon_texte`.

Consignes

Attention, dans cet exercice, il n'y a rien à afficher, donc vous ne ferez aucun appel à la fonction `print`.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Pour tester votre code, UpyLaB va l'exécuter en ajoutant des instructions lui permettant de vérifier que les variables attendues existent et sont bien affectées des valeurs attendues.
Si vous souhaitez tester votre code **dans votre IDE (Thonny ou PyCharm par exemple)**, pensez à ajouter les instructions :

```
print("x =", x)
print("produit =", produit)
print("div_entiere =", div_entiere)
...
```

etc, pour visualiser le contenu de vos variables.
— Veillez à bien respecter les noms de variables attendus.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 2.1*.

2.5.6 Exercice UpyLaB 2.2 : Vert

Le but de cet exercice est de vérifier que vous savez lire des données en entrée avec la fonction `input`, les affecter à des variables et imprimer (on dit aussi afficher) une valeur grâce à la fonction `print`.

Écrire un programme qui imprime (donc grâce à la fonction `print`) la moyenne arithmétique de deux nombres de type `float` lus en entrée (c'est-à-dire grâce à des appels à la fonction `input`) .

On rappelle que la moyenne arithmétique de deux nombres a et b est égale à $\frac{a+b}{2}$

Exemple 1

Avec les données lues suivantes :

```
2.0
3.0
```

le résultat à imprimer vaudra :

```
2.5
```

Exemple 2

Avec les données lues suivantes :

```
4.2
3.8
```

le résultat à imprimer vaudra :

```
4.0
```

Consignes

Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.

En particulier, il ne faut rien écrire à l'intérieur des appels à input (`float(input())` et non `float(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
- Portez attention aux règles de priorité des opérateurs arithmétiques.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.2*.

2.5.7 Exercice UpyLaB 2.3 : Vert

Le but de cet exercice est de vous familiariser avec la lecture (`input()`) de données et l'impression (`print()`) de résultats.

Une méthode pour trouver le quatrième terme parmi quatre termes ayant un même rapport de proportion $\frac{a}{b} = \frac{c}{d}$ lorsque trois de ces termes sont connus repose sur l'égalité des produits en croix.

Elle utilise le fait que le produit des premier et quatrième termes est égal au produit du second et du troisième : $a.d = b.c$ et donc $d = \frac{b.c}{a}$

Exemple : si chacun mange autant de chocolat et que pour 4 personnes il en faut 100 grammes, pour 7 personnes il en faudra donc d tel que $\frac{4}{100} = \frac{7}{d}$

D'où $d = \frac{7.100}{4}$ grammes = 175 grammes.

Écrire un programme qui lit des valeurs de type float pour a, b et c et qui affiche la valeur de d vérifiant l'égalité $\frac{a}{b} = \frac{c}{d}$.

Exemple 1

Avec les données lues suivantes :

```
4.0
100.0
7.0
```

le résultat à imprimer vaudra :

```
175.0
```

Exemple 2

Avec les données lues suivantes :

```
3.5
0.5
8.0
```

le résultat à imprimer vaudra :

```
1.1428571428571428
```

Remarque : Du fait du manque de précision dans les calculs avec les nombres de type float, votre résultat pourra légèrement différer de celui indiqué ci-dessus. Ce n'est pas un problème, car UpyLaB acceptera toute réponse suffisamment proche du résultat attendu, avec une tolérance d'environ 1.0e-5.

Consignes

Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.

En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 2.3*.

2.6 Quelques fonctions prédéfinies, les modules math et turtle

2.6.1 Exemples d'utilisations de fonctions prédéfinies, et des modules math et turtle

PRÉSENTATION

Les capsules suivantes montrent que Python fournit au programmeur des modules avec des constantes et des fonctions prédéfinies, c'est-à-dire déjà écrites, qu'il peut utiliser à sa guise soit directement, soit en les important.

VIDÉO SUR LE MODULE MATH

Note : Voir la vidéo de la section 2.6.1 : Le module math

VIDÉO SUR LE MODULE TURTLE

Note : Voir la vidéo de la section 2.6.1 : Le module turtle

CONTENU DES VIDÉOS

Les deux vidéos précédentes ont introduit les modules Python `math` et `turtle`.

Nous donnons ici les scripts qui y sont utilisés pour la présentation suivis d'un résumé des explications données s'y rapportant.

Scripts et consoles utilisés dans les vidéos

Le module math

```
""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = 3.14 * 2 * rayon  # calcul de la circonférence
```

(suite sur la page suivante)

(suite de la page précédente)

```
aire = 3.14 * rayon **  2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)
```

```
""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

pi = 3.14159

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = pi * 2 * rayon  # calcul de la circonférence
aire = pi * rayon **  2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)
```

```
""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

import math

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = math.pi * 2 * rayon  # calcul de la circonférence
aire = math.pi * rayon **  2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)
```

```
>>> import math
>>> help(math)
...
>>>  dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
↪'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
↪'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
↪'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
↪'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
↪'tanh', 'tau', 'trunc']
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> math.cos(0.5)
0.8775825618903728
>>> math.cos(math.pi)
-1.0
>>> from math import cos, pi
>>> cos(pi)
-1.0
```

```
""" Auteur: Sébastien Hoarau
    Date : Juin 2018
    But du programme :
    Le programme suivant calcule la circonférence
    et l'aire d'un disque dont le rayon est donné
    en input
    Entrée: rayon : le rayon du disque
    Sorties: la circonférence du disque, l'aire du disque
"""

from math import pi

#lecture du rayon :
rayon = float(input("Veuillez donner le rayon : "))
circ = pi * 2 * rayon  # calcul de la circonférence
aire = pi * rayon **  2 # calcul de l'aire

#Affichage des résultats
print("Circonférence :", circ)
print("Aire          :", aire)
```

Le module turtle

```
import turtle
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
turtle.forward(100)
turtle.left(90)
```

```
import turtle

turtle.up()
turtle.goto(-150,-150)
turtle.down()
turtle.color("blue")
turtle.begin_fill()
turtle.goto(150,-150)
turtle.goto(150,150)
turtle.goto(-150,150)
turtle.goto(-150,-150)
turtle.end_fill()
turtle.done() # turtle.mainloop()
```

La vidéo en bref

Cette section montre que Python fournit au programmeur des modules avec des constantes et des fonctions prédéfinies, c'est-à-dire déjà écrites, qu'il peut utiliser à sa guise soit directement, soit en les important.

Le module math

Au début du script, après le *docstring initial*, on peut rajouter la ligne de code

```
import math
```

Dans ce cas, pour utiliser pi, je dois spécifier que c'est l'attribut du module math en tapant `math.pi` qui me donne la valeur approximative de pi (avec les 15 ou 16 premiers chiffres corrects vu la précision possible).

Si je n'ai besoin que de pi et cos du module math par exemple, je peux préciser et écrire

```
from math import cos, pi
```

Et alors `cos(pi)`, qui appelle la fonction cos() avec pi en argument, et qui donc demande la valeur du cosinus de pi, est bien compris par l'interpréteur Python.

Le module turtle

Le module `turtle` permet de dessiner des figures de façon très simple. `turtle` peut être vue comme une tortue qui porte une plume. Quand elle se déplace, soit la plume est descendue, ce qui est le cas au début, et dans ce cas elle trace une ligne lors de ses déplacements, soit la plume est relevée et dans ce cas, elle ne trace rien.

Une liste de commandes `turtle`, les plus communes dont nous allons nous servir avec leur explication, est donnée plus loin dans le cours.

Pour laisser la fenêtre ouverte à la fin du programme jusqu'à ce que l'utilisateur décide de la fermer en cliquant sur le bouton rouge de fermeture de la fenêtre où turtle a dessiné, il suffit d'ajouter comme dernière instruction du code : `turtle.done()` ou bien `turtle.mainloop()`.

2.6.2 Quelques fonctions prédéfinies et fonctions turtle très utilisées

FONCTIONS PRÉDÉFINIES

Dans la vidéo précédente, nous avons déjà utilisé certaines fonctions prédéfinies ; en voici une liste non exhaustive qui vous sera sûrement bien utile pour résoudre des exercices dans le cadre de ce cours ou de projets futurs :

- `abs(x)`
- `dir(x)`
- `divmod(x, y)`
- `float(x)`
- `help(x)`
- `input()`
- `int(x)`
- `max(a, b,...)`
- `min(a, b,...)`
- `print()`
- `round(x, y)`
- `sum(a)`
- `type(x)`

N'hésitez pas à vous les approprier en les manipulant dans une console Python avec différents arguments donnés à la fonction appelée, à utiliser la fonction help() ou même à aller voir dans la documentation python3 pour bien comprendre ce que ces fonctions font.

Par exemple pour comprendre comment fonctionne la fonction `round()`, tapez dans une console PyCharm `help(round)` (et si l'anglais n'est pas votre fort, n'oubliez pas que sur internet des traducteurs automatiques en ligne peuvent vous aider).

Par exemple tapez dans la console

```
>>> help(divmod)
```

pour savoir l'effet de cette fonction. L'explication utilise le mot invariant : cela signifie un fait toujours vrai.

isinstance

Si l'on veut tester si une valeur `ma_variable` est de type `int` par exemple, la fonction booléenne prédéfinie `isinstance` peut être utilisée avec `isinstance(ma_variable, int)` plutôt que `type(ma_variable) is int`. La différence entre les deux façons de faire n'est visible qu'en programmation orientée-objet (si dans une console python vous tapez `help(isinstance)`, il est dit que `isinstance(obj, class)` teste si `obj` est une instance d'une classe ou une sous-classe de `class`). Dans le cadre de ce cours, comme la notion de sous-classe d'une classe n'a pas été vue et n'est pas utilisée, les deux utilisations sont équivalentes.

Aide-mémoire : Pour vous aider tout au long de ce cours, nous avons confectionné un aide-mémoire, qui explique succinctement l'effet des fonctions et méthodes prédéfinies les plus couramment utilisées.

L'aide-mémoire est disponible à l'adresse https://www.fun-mooc.fr/asset-v1:ulb+44013+session05+type@asset+block@aide-memoire-Python.pdf

Nous vous conseillons même de l'imprimer, et de l'avoir sous la main quand vous programmez. Notez que jusqu'à présent nous n'avons étudié qu'une petite partie du contenu de cet aide-mémoire. Soyez patient, la suite arrive !

Module math

Le module math est particulièrement utile pour écrire des scripts résolvant des problèmes mathématiques. Le but de ce cours n'est sûrement pas de connaître le contenu de modules comme math par coeur. Par contre, il est important lorsque nous écrivons un programme Python de savoir trouver de l'information complémentaire. Une première exploration du contenu d'un module, comme le module math, consiste à ouvrir une console Python et à exécuter :

```
>>> help('math')
```

Soyez curieux et explorez ce module ou le module `turtle` par exemple !

Module turtle

Comme annoncé dans la vidéo précédente, dans ce cours, nous allons également utiliser le module `turtle` qui va nous permettre de dessiner des figures de façon très simple sur une fenêtre graphique.

Comme les exemples dans la vidéo le montrent, le module `turtle` utilise deux façons pour se déplacer : soit en demandant avec les verbes `forward`, `backward`, `left`, `right` d'avancer (forward), de reculer (backward), de tourner à gauche (left) ou à droite (right), soit en spécifiant, avec des `goto`, les coordonnées `(x, y)` dans le plan où la tortue doit se déplacer.

— la valeur de l'"*abscisse* `x` donnant la position gauche / droite (plus la valeur est négative, plus elle est à gauche de la fenêtre, plus elle est positive, plus elle est à droite) ;
— la valeur de l'"*ordonnée* `y` donnant la position bas / haut (plus la valeur est négative, plus elle est en bas de la fenêtre, plus elle est positive, plus elle est en haut).

Initialement, la tortue se trouve au centre de la fenêtre, en coordonnée `(0, 0)`. Utiliser dans des goto des valeurs trop petites (par exemple -400) ou trop grandes (par exemple 400) fait sortir la tortue de la fenêtre.

Voici une liste de commandes `turtle`, les plus communes, dont nous allons nous servir :

Commande	Effet
reset()	On efface tout et on recommence
goto(x, y)	Aller à l'endroit de coordonnées x, y
forward(distance)	Avancer d'une distance donnée
backward(distance)	Reculer
up()	Relever le crayon (pour pouvoir avancer sans dessiner)
down()	Abaisser le crayon (pour recommencer à dessiner)
color(couleur)	couleur peut être `'red'`, `'blue'`, etc.
color(bord, zone)	couleur du bord et de la zone
left(angle)	Tourner à gauche d'un angle donné (exprimé en degrés)
right(angle)	Tourner à droite
width(épaisseur)	Choisir l'épaisseur du tracé
begin_fill()	Début de la zone fermée à colorier
end_fill()	Fin de la zone fermée à colorier
hideturtle()	Cache la tortue
showturtle()	Montre la tortue
write(texte)	Ecrit texte
done() ou mainloop()	Attend que l'utilisateur ferme la fenêtre

Nous vous invitons à ouvrir une console Python et à essayer ces commandes.

TESTONS LES FONCTIONS TURTLE

Pour bien visualiser le rôle des fonctions couramment utilisées par `turtle`, créez un script dans Thonny ou PyCharm qui contient le code ci-dessous et exécutez-le (Run). Afin de vous aider à analyser l'effet de chaque instruction, nous vous suggérons de commenter chaque ligne en expliquant ce qu'elle fait et en ayant découvert quelle figure est dessinée par ce code.

Note : Attention, surtout n'appelez pas votre script turtle.py qui est le nom utilisé pour le module `turtle` lui même. Si vous faites cela, l'import du module turtle ne sera pas réalisé.

```
import turtle
turtle.up()
turtle.shape('turtle')
turtle.goto(-80,0)
turtle.color('blue')
turtle.down()
turtle.begin_fill()
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.forward(300)
turtle.right(144)
turtle.end_fill()
turtle.hideturtle()
turtle.done()
```

PROPOSITION DE SOLUTION

```
""" auteur: Thierry Massart
   date: 7 décembre 2017
   Trace avec turtle une étoile dont les extrémités sont bleues
"""
import turtle           # importation du module turtle
turtle.up()             # tant que la tortue est en mode "up",
                        # son déplacement ne trace rien
turtle.shape('turtle') # change la forme de la tortue (en tortue)
turtle.goto(-80,0)      # la tortue se place en coordonnées (-80, 0)
                        # (-80 pour l'axe des "x" et 0 l'axe des "y")
turtle.color('blue')    # la tortue est bleue
turtle.down()           # tant que la tortue est "down",
                        # elle tracera la ligne de ses déplacements
turtle.begin_fill()     # va remplir l'intérieur de ce qui est tracé entre
                        # maintenant et le turtle.end_fill() ultérieur
turtle.forward(300)     # la tortue avance de 300 (à droite)
turtle.right(144)       # la tortue effectue une rotation de 144° à droite
turtle.forward(300)     # avance dans la nouvelle direction
turtle.right(144)       # rotation
turtle.forward(300)     # avance
turtle.right(144)       # ...
turtle.forward(300)     #
turtle.right(144)       #
turtle.forward(300)     #
turtle.right(144)       #
turtle.end_fill()       # remplit ce qui a été tracé entre le begin_fill
                        # et cette instruction
turtle.hideturtle()     # cache la tortue
turtle.done()           # laisse l'utilisateur fermer la fenêtre
```

2.6.3 Pavé hexagonale

PAVÉ HEXAGONAL

Lancez-vous avec `turtle` : faites fonctionner votre imagination pour dessiner des figures. Vous pouvez aussi dessiner des polygones réguliers à 3, 5, 7 . . . côtés.

Si vous avez besoin de vous rafraîchir la mémoire sur certaines notions de vos cours de mathématiques, une rapide recherche sur le Web vous permettra de (re)découvrir que pour dessiner un polygone à n côtés (par exemple n = 5), l'angle de la rotation à réaliser entre deux côtés est de 360°/n. Si n vaut 5 cela donne donc 360°/5 = 72°.

Si l'on veut dessiner des étoiles à n branches et en supposant n impair (pour n pair, c'est un peu plus délicat) au moins égal à 5, l'angle intérieur sera de (n-1)*180°/n (par exemple 144° pour n valant 5 comme donné dans mon code précédent).

Un exemple de pavé hexagonal, est la composition de trois losanges de couleurs différentes comme ici en noir, bleu et rouge. Visionnez la vidéo pour comprendre la séquence réalisée par le code.

Note : Voir la vidéo de la section 2.6.3 : dessin d'un pavé hexagonal avec turtle

---

Pourquoi ne pas essayer : dessinez avec turtle un tel pavage en partant du point (0, 0) qui est la position initiale de la tortue, avec des lignes de longueur 100 par exemple.

- D'abord en utilisant en particulier les instructions `turtle.forward()`, `turtle.left()` et `turtle.right()` (Un petit conseil : utilisez des angles de 60° ou de 120° pour faire tourner votre tortue ; voir la figure « Géométrie d'un hexagone »)
- Ensuite, encore mieux, utilisez l'instruction `turtle.goto()` pour vos déplacements en calculant les abscisses et ordonnées des points extrémités où doit se déplacer la tortue ; voir les deux figures « Géométrie d'un hexagone » et « Rappel sur

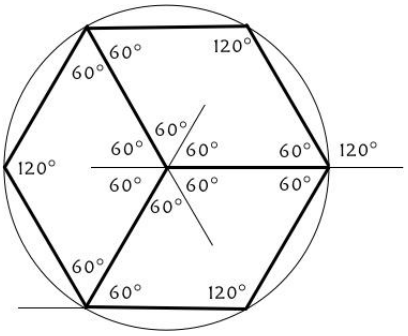


Fig. 2.1 – Géométrie d'un hexagone

les sinus et cosinus ». Une des figures vous rappelle comment vous pouvez calculer ces coordonnées avec les fonctions sinus et cosinus (`math.sin` et `math.cos` après avoir importé le module `math`) : dans un espace à 2 dimensions, pour un cercle de centre (0,0) et de rayon 1, les coordonnées du point sur le cercle ayant un angle `alpha` avec l'axe des x (le point en rouge) est donné par `(cosinus(alpha), sinus(alpha))`. Dans la figure « Rappel sur les sinus et cosinus », l'angle `alpha` vaut 60 degrés soit pi / 3 radians.

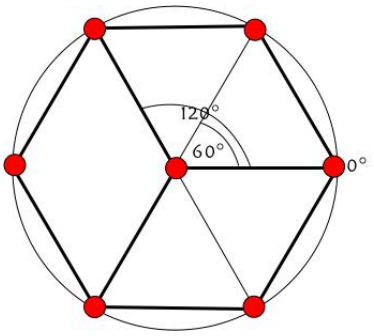


Fig. 2.2 – Géométrie d'un hexagone

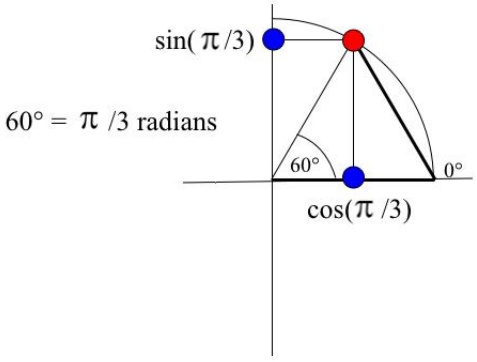


Fig. 2.3 – Rappel sur les sinus et cosinus

Avant de réaliser le pavage hexagonal nous vous demandons de réaliser l'exercice 2.4 du module 2 d'UpyLaB donné ci-après. La réussite de cet exercice vous permettra de réaliser plus facilement le pavage demandé ci-dessus.

Comme l'exercice contient un bagage mathématique en trigonométrie qui n'est pas l'objet de ce cours, nous proposons à ceux qui n'ont pas ou plus la connaissance des notions de sinus et cosinus d'avoir accès à une solution en bas de l'exercice UpyLaB 2.4. Notons que de ce fait, l'exercice n'est pas noté.

2.6.4 Exercice UpyLaB 2.4 - Non noté : Bleu

Écrire un programme qui lit une longueur `long` de type float strictement positive, et qui affiche les valeurs x y des coordonnées (x, y) des sommets de l'hexagone de centre (0,0) et de rayon `long`

Chaque couple de coordonnées sera affiché sur une ligne différente, en commençant par le point à 0°, puis par le point à 60°, puis 120° … jusqu'au 6ème point.

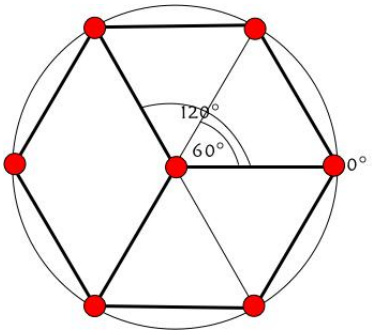


Fig. 2.4 – Géométrie d'un hexagone

Exemple 1

Avec la donnée lue suivante :

```
100.0
```

le résultat à imprimer vaudra (approximativement) :

```
100.0 0.0
50.000000000000014 86.60254037844386
-49.99999999999998 86.60254037844388
-100.0 1.2246467991473532e-14
-50.00000000000004 -86.60254037844383
50.000000000000014 -86.60254037844386
```

Exemple 2

Avec la donnée lue suivante :

```
5.5
```

le résultat à imprimer vaudra (approximativement) :

```
5.5 0.0
2.7500000000000004 4.763139720814412
-2.7499999999999987 4.763139720814413
-5.5 6.735557395310443e-16
-2.7500000000000027 -4.7631397208144115
2.7500000000000004 -4.763139720814412
```

Consignes

— Pour les calculs utilisez la valeur `pi` et les fonctions `sin` et `cos` du module math, comme indiqué sur la figure « Coordonnées d'un sommet de l'hexagone ».

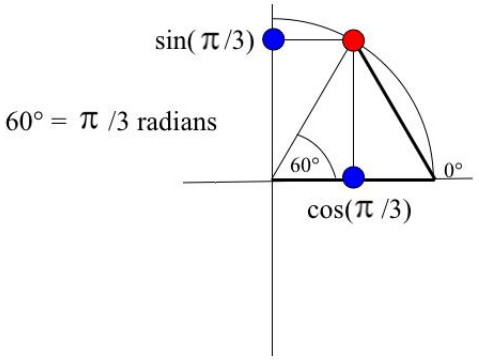


Fig. 2.5 – Coordonnées d'un sommet de l'hexagone

— Notez qu'il n'est pas demandé de tester si la valeur lue est bien strictement positive ; nous vous assurons que ce sera le cas pour les valeurs passées au programme dans les tests d'UpyLaB.
— Il ne vous est **pas** demandé de tracer l'hexagone, mais uniquement d'afficher des valeurs. Notez qu'UpyLaB ne supporte pas le module `turtle` et donc ne l'importez pas dans votre code pour cet exercice !
— Vous constaterez que l'utilisation des fonctions `sin()` et `cos()` et les erreurs d'arrondis et de troncature durant les calculs induiront que les résultats peuvent être un peu différents du résultat théorique ;
Pour l'exemple 1 plus haut, les résulats plus précis devraient être :

```
 100.0    0.0
 50.0     86.6025403784438
-50       86.6025403784438
-100.0    0.0
-50.0    -86.6025403784438
 50.0    -86.6025403784438
```

où 86.6025403784438 est une valeur approchée de la valeur de $100.\frac{\sqrt{3}}{2}$.
UpyLaB laisse une certaine latitude (ici de l'ordre de $1.0e-5$).
— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : ")` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res))` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test un nombre différent en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes.
— Veillez à ce que votre affichage corresponde bien à ce qui est attendu ; `(1.0, 0.0)` n'est pas la même chose que `1.0 0.0`. Utilisez deux variables distinctes pour chacune des coordonnées et imprimez-les grâce à l'instruction `print(x, y)`.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 2.4*. Vous y trouverez en particulier quelques indices sur les calculs à effectuer.

Et si vraiment, vous ne trouvez pas

Vous n'arrivez pas à réaliser l'exercice. Voici une solution pour le résoudre :

```
from math import pi, sin, cos

long = float(input())
print(long * cos(0), long * sin(0))
print(long * cos(pi / 3), long * sin(pi / 3))
print(long * cos(pi * 2 / 3), long * sin(pi * 2 / 3))
print(-long, 0.0)
print(long * cos(4 / 3 * pi), long * sin(4 / 3 * pi))
print(long * cos(5 / 3 * pi), long * sin(5 / 3 * pi))
```

2.7 Pour terminer ce module

2.7.1 Stockage des valeurs et caractères d'échappement

PETITES REMARQUES SUR LES TROIS TYPES DÉJÀ VUS

Dans ce cours, nous ne voulons pas vous donner d'informations inutiles pour votre apprentissage. Par exemple, il est à ce stade inutile de trop détailler comment l'interpréteur Python réalise son travail. Il est pourtant important d'avoir certains éléments qui vont vous permettre de mieux comprendre le pourquoi des choses, et ainsi de savoir à quoi il faut faire attention dans vos codes.

Ainsi, pourquoi Python distingue-t-il les valeurs entières et fractionnaires ?

La réponse est dans le stockage en mémoire de l'ordinateur.

Entiers (int)

Tant que l'interpréteur Python a de la place mémoire disponible, il stocke des entiers aussi grands que demandé.

Si l'on demande à l'interpréteur Python dans une console interactive

```
>>> 3**100
>>> 3**1000
>>> 3**10000
```

les calculs peuvent prendre du temps mais vous pouvez vérifier que l'interpréteur donne le résultat.

Par contre, les valeurs entières ne stockent bien évidemment pas les parties fractionnaires ; ainsi

```
>>> 1 // 3
```

ou même

```
>>> 1 // 3 * 3
```

donnent la valeur 0.

Fractionnaires (float)

Pour les valeurs `float`, l'interpréteur fait des calculs et encode les valeurs dans un mot mémoire généralement de 64 bits, avec la meilleure précision qu'il peut et en retenant également le signe et une partie exposant pour pouvoir représenter des petits ou des grands nombres (grosso modo jusqu'environ 10e-300 pour le plus petit nombre strictement positif et 10e300 pour le plus grand).

Pour comprendre comment cela se passe, un parallèle peut être fait avec la représentation décimale des nombres fractionnaires :

si on doit représenter 1/3 en nombre décimal avec un nombre fixé, par exemple 5, de chiffres après le point décimal, cela donne 0.33333 et on perd de la précision.

La représentation de nombres float en Python est similaire sauf que tout est représenté en binaire, c'est-à-dire avec comme seuls chiffres des 0 et des 1.

Nous vous invitons à tester cela vous-même dans une console en tapant par exemple :

```
>>> 1.0e308
>>> 1.0e309
>>> 1.0e-323
>>> 1.0e-324
>>> 1.00000000000000001
```

pour voir ce que l'interpréteur donne comme valeurs.

En général, on perd donc de la précision dans les calculs et parfois les calculs ne peuvent être simplement réalisés avec les valeurs manipulées en tant que valeurs de type float.

Par exemple, à la place de 1.0e309, l'interpréteur affiche la valeur `inf` : il s'agit de l'infini qui marque que la valeur est trop grande pour être manipulée avec des float. Si vous désirez manipuler de telles valeurs dans vos codes Python, il faudra trouver des types plus étendus pour le faire. C'est possible mais sort du cadre de ce cours.

Pour ceux qui veulent faire des calculs scientifiques précis et désirent une explication plus complète ainsi que des alternatives pour faire mieux, allez voir sur le site python.org, dans la documentation Python 3 et cherchez "Floating Point Arithmetic : Issues and Limitations".

Chaînes de caractères avec caractères d'échappement

Il se peut que nous voulions imprimer du texte avec des tabulations, passages à la ligne ou autres caractères spéciaux comme des quotes simple ou doubles à l'intérieur du texte.

Pour cela, on peut utiliser le caractère d'échappement (appelé "escape" en anglais).

Par exemple

```
print('\n\nBonjour\n\n')
```

demande d'imprimer le texte qui contient au début 2 passages à la ligne suivi du texte Bonjour et terminé par à nouveau 2 passages à la ligne (soit 11 caractères au total qui seront imprimés sur 4 lignes).

Le tableau suivant donne quelques caractères spéciaux dénotés grâce au caractère d'échappement. Notez que de ce fait, pour spécifier qu'un caractère antislash fait partie du texte, il faut mettre deux antislashs : la première exprimant que ce qui suit est le caractère antislash lui-même.

```
print('\\')
```

imprime juste un seul caractère antislash.

Notez que si le caractère qui suit l'antislash dans un texte n'est pas reconnu comme caractère spécial, l'antislash est interprétée littéralement par l'interpréteur python.

Ainsi `print('\B')` imprime bien les deux caractères `\B`.

`\'`	La simple quote (')
`\"`	La double quote (")
`\n`	Le passage à la ligne ASCII
`\t`	La tabulation horizontale
`\\`	L'antislash (backslash \)

N'hésitez pas à tester ces caractères par vous-même pour être sûr que vous les maîtrisez.

Note : Chaîne de caractères multiligne

Nous avons vu qu'au début de nos scripts, il était recommandé de mettre un commentaire multiligne donnant des informations sur ce que fait ce script, son auteur, la date de création et éventuellement de dernière modification.

Un commentaire multiligne est en fait une chaîne de caractères multiligne qui peut être manipulée comme n'importe quelle autre chaine de caractères. Sa particularité est qu'il peut s'étaler sur plusieurs lignes ; chaque passage à la ligne correspondant au caractère dénoté `\n`.

Par exemple, le code

```
mon_texte =  '\n\nBonjour\n\n'
```

peut être écrit :

```
mon_texte =  """

Bonjour

"""
```

Dans ce cas, faites attention à l'indentation puisque ici, tout ce qui est entre les trois doubles quotes fait partie du texte.

2.7.2 Les opérateurs d'assignation et de mise à jour

COMPLÉMENT SUR L'ASSIGNATION

Pour être précis l'interpréteur Python manipule des objets contenant les valeurs et des variables qui sont des noms pour les objets. Notons que l'instruction d"assignation multiple

```
x = y = 3
```

crée un objet de type entier contenant la valeur 3 ; `x` et `y` sont deux noms donnés à cet objet.

Si juste après, l'interpréteur exécute

```
x = 4
```

après cette instruction, la variable `x` aura changé de valeur et vaudra 4 (`x` désigne la valeur 4), tandis que `y` reste inchangée et a toujours la valeur 3.

Les opérateurs de mise à jour

Python permet d'écrire de façon courte des opérations qui consistent à prendre la valeur d'une variable et de lui appliquer une opération avec une valeur comme deuxième opérande.

L'exemple le plus courant est l"incrémentation :

supposons que `x` soit égal à 5 (par exemple après l'assignation `x = 5`)

```
x += 1
```

incrémente `x` de 1, lui donnant la valeur 6.

Attention

```
x = 5
x += 5 * 7
```

Le code ci-dessus assigne `5` à la variable `x`, ensuite évalue `5 * 7` avant d'incrémenter `x` du résultat. C'est donc l'équivalent de `x = x + (5 * 7)`.

Ce principe peut être appliqué avec tous les opérateurs arithmétiques (`+=`, `-=`, `*=`, `/=`, `//=`, `**=`, `%=`).

2.7.3 On passe à la pratique autonome !

MISE EN PRATIQUE AVEC UPYLAB

Comme déjà annoncé, on ne peut devenir un bon codeur sans la mise en pratique répétée des concepts vus.

C'est le rôle d'UpyLaB de vous obliger à réaliser cette pratique.

Nous vous demandons de réaliser l'ensemble des exercices du module 2 qu'il vous reste à faire et qui sont donnés dans les pages suivantes de cette section.

En cas de difficultés, nous vous rappelons ces quelques conseils :

1. Relire l'énoncé et les consignes associés à l'exercice ainsi que les éventuels conseils.
2. Lire la FAQ générale à propos des exercices UpyLaB.
3. Lire, si elle existe, la FAQ spécifique à l'exercice réalisé.
4. Si rien de tout cela ne fonctionne, poser votre question sur le Forum dédié aux exercices, en étant le plus précis possible, *sans mettre votre code complet ni substantiel* mais en mettant des copies d'écran des résultats fournis par UpyLaB.

2.7.4 Exercice UpyLaB 2.5 : Vert

Énoncé

Le but de cet exercice est de vous familiariser avec la syntaxe Python pour écrire des expressions arithmétiques simples et avec l'instruction `print` qui affiche (on dit aussi imprime) des valeurs à l'écran.

Écrire un programme qui lit deux valeurs entières x et y strictement positives suivies de deux valeurs réelles (float) z et t, et qui affiche les valeurs des expressions suivantes, chacune sur une nouvelle ligne :

— $x - y$
— $x + z$
— $z + t$
— $x.z$ (produit de x et de z)
— $\frac{x}{2}$
— $\frac{x}{y+1}$
— $\frac{(x+y).z}{4.x}$
— $x^{-\frac{1}{2}}$ (x exposant $-\frac{1}{2}$)

Exemple

Avec les données lues suivantes :

```
2
1
3.0
3.5
```

le résultat à imprimer vaudra (approximativement pour la dernière valeur) :

```
1
5.0
6.5
6.0
1.0
1.0
1.125
0.7071067811865476
```

Consignes

— Il n'est pas demandé de tester si les valeurs de `x` et de `y` sont bien strictement positives.
— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 2.5*.

2.7.5 Exercice UpyLaB 2.6 : Vert

Énoncé

Le but de cet exercice est de vous familiariser avec l'impression de certains caractères particuliers comme les simples et double quotes (apostrophes), l'anti-slash » (appelée également la barre oblique inversée \), ...

Écrire un programme affichant les quatre lignes suivantes :

```
Hello World
Aujourd'hui
C'est "Dommage !"
Hum \o/
```

Consignes

Veillez à ce que votre programme n'affiche rien de plus que ce qui est attendu : pas d'espace en fin de ligne, pas de ligne vide supplémentaire, ..., et faites attention aux détails : respectez la ponctuation (les simples et doubles quotes sont droites), le nombre d'espaces entre les mots...

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si votre code n'est pas accepté et qu'il semble pourtant afficher ce qui est attendu, c'est qu'il peut y avoir des espaces superflues en fin de ligne. Attention en particulier à la fonction `print`, qui ajoute par défaut une espace entre ses arguments s'il y en a plusieurs. Une solution consiste à modifier l'argument nommé `sep` de cette fonction, en lui attribuant par exemple la valeur chaîne vide (`sep = ""`), ou encore à utiliser plusieurs appels à cette fonction.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 2.6*.

2.7.6 Exercice UpyLaB 2.7 : Bleu

Écrire un programme qui imprime la valeur du volume d'une sphère de rayon `r`, float lu en entrée.

On rappelle que le volume d'une sphère de rayon r est donné par la formule : $\frac{4}{3}\pi r^3$

Exemple 1

Avec la donnée lue suivante :

```
1.0
```

le résultat à imprimer vaudra (approximativement) :

```
4.1887902047863905
```

Exemple 2

Avec la donnée lue suivante :

```
0.5
```

le résultat à imprimer vaudra (approximativement) :

```
0.5235987755982988
```

Consignes

- Il n'est pas demandé de tester si la valeur lue en entrée est bien positive ou nulle.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
- Pour rappel, en Python, l'opérateur exposant est `**`. Ainsi, `2 ** 3` vaut `8` soit `2 * 2 * 2`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 2.7*.

2.8 Quiz de fin et bilan du module

2.8.1 Quiz de fin de module

MINI QUIZ

Note : Voir le quiz de la section 2.8.1

BILAN EN BREF

Note : Voir la vidéo de la section 2.8.2 : Bilan du module 2

BILAN DU MODULE

Nous voici à la fin de ce module. Nous avons déjà vu beaucoup de concepts de base Python 3 dont voici les principaux :

— Nous avons vu que l'interpréteur Python 3 peut être utilisé en mode interactif ou en mode script en particulier dans l'environnement PyCharm.
— Nous avons vu qu'un code Python peut utiliser des valeurs et expressions entières (int), fractionnaires (float) ou chaînes de caractères (str pour string), la fonction prédéfinie `type(x)` donnant le type de `x`.
— Votre code peut être déclaré incorrect parce qu'il contient des erreurs syntaxiques.
— Nous avons également vu que Python peut se créer des variables ayant des valeurs grâce à l'instruction d'"*assignation* (appelée également *affectation*).
— Les instructions `input` et `print` permettent au script Python de "recevoir des données" de l'utilisateur et d" "imprimer des résultats".
— Pour avoir un code plus lisible pour le programmeur, des commentaires simples et multilignes peuvent être écrits dans le code.
— Python offre des fonctions prédéfinies (`type()`, `int()`, `float()`, `dir()` et `help()` et `max()`, `round()`, `divmod()`...), ainsi que des modules comme `math` et `turtle`, utilisables via le verbe `import`.
— Nous avons vu que les diagrammes d'état expliquent de façon graphique comment fonctionne un code Python et que l'outil Python Tutor (accessible via la page web pythontutor.com) permet de visualiser ces diagrammes d'état en exécutant pas à pas les petits scripts Python fournis.
— Nous avons également illustré tous ces concepts avec de nombreux exemples et la réalisation d'exercices supervisés ou réalisés de façon autonome avec UpyLaB.

CHAPITRE 3

Les instructions : tous vos désirs sont des ordres

3.1 Presque toutes les instructions au menu

3.1.1 Présentation du menu de ce module

LES INSTRUCTIONS PYTHON

La vidéo qui suit présente le menu du présent module : l'étude des instructions de contrôle de flux Python. Elles sont appelées instructions de contrôle de flux parce qu'elles permettent de rompre la simple séquence d'exécution comme ce que nous avons vu jusqu'à présent. Nous verrons que ces instructions sont essentielles pour rédiger du code Python.

MENU DE CE MODULE

Note : Voir la vidéo de la section 3.1.1 : Menu du module 3

3.2 L'instruction conditionnelle if : fais pas « si », fais pas ça !

3.2.1 L'instruction if

INTRODUCTION À L'INSTRUCTION CONDITIONNELLE IF

La présente section vous explique tout ce qui vous sera nécessaire sur l'instruction conditionnelle `if`.

Cette vidéo introduit l'instruction `if` qui permet d'exécuter certaines instructions uniquement si une condition est vérifiée.

Comme le traitement va dépendre d'une condition, nous parlerons de l'instruction conditionnelle `if`.

Nous partons d'un exemple simple avec seulement le mot-clé `if` pour expliquer le fonctionnement de cette instruction et petit à petit, nous ajoutons des éléments, en particulier qui utilisent les mots-clés `elif` et `else`, pour montrer différentes possibilités.

PRÉSENTATION DE L'INSTRUCTION IF

Note : Voir la vidéo de la section 3.2.1 : L'instruction if

CONTENU DE LA VIDÉO

La vidéo précédente présente l'instruction conditionnelle `if` et ses formats possibles : sans parties `else` et `elif`, avec une partie `else` et avec une partie `elif`.

Scripts et code sur la console réalisés dans la vidéo

```
maximum = 0
releve = int(input())
if releve > maximum :
    maximum = releve
    print("Nous avons un nouveau record")
print("Maximum retenu :", maximum)
```

```
>>> 3 < 5
True
>>> 5 < 3
False
>>> type(False)
<class 'bool'>
>>> type(True)
<class 'bool'>
>>> type(3<5)
<class 'bool'>
>>> x = int(input())
>? 5
>>> if x < 0:
...     print("ok")
...

>>>
```

```
maximum = 10
releve = int(input())
if releve > maximum :
    maximum = releve
    print("Nous avons un nouveau record")
else:
    print("Pas de nouveau record")
print("Maximum retenu :", maximum)
```

```
maximum = 10
releve = int(input())
if releve == 0:
    print("Pas de pluie aujourd'hui")
elif releve > maximum :
    maximum = releve
    print("Nous avons un nouveau record")
else:
    print("Pas de nouveau record")
print("Maximum retenu :", maximum)
```

La vidéo en bref

L'instruction conditionnelle `if` est présentée grâce à des scripts. Sa forme la plus simple est :

```
if condition:
    instructions
```

où la condition peut par exemple être une comparaison entre deux valeurs en utilisant un opérateur relationnel tel que :
- *est strictement inférieur* < ou
- *est strictement supérieur* >.

Cette comparaison donne une valeur de type *booléenne* vraie (`True`) ou fausse (`False`).

L'indentation permet d'identifier les instructions dans le `if`.

Un `else` peut être ajouté à la fin de l'instruction `if` et une ou plusieurs parties `elif` peuvent être également ajoutées comme montré dans la vidéo.

L'instruction `if` avec parties `elif` et `else` a la forme :

```
if condition:
    instructions
elif condition:
    instructions
elif condition:
    instructions
...
else:
    instructions
```

Dans tout les cas, au maximum une seule « branche » est exécutée, et elle correspond au premier test évalué à vrai soit au niveau du `if` soit au niveau d'un `elif` et sinon, au niveau du `else` s'il existe. Ensuite l'interpréteur Python passe à l'instruction suivante.

Si tous les tests du `if` et `elif` sont évalués à faux et qu'il n'y a pas de partie `else`, aucune « branche » du `if` ne sera exécutée.

3.2.2 Les opérateurs relationnels

LES OPÉRATEURS RELATIONNELS

Nous avons vu qu'un élément clé de l'instruction `if` est la condition associée, dont la valeur, vraie (`True`) ou fausse (`False`), est souvent le résultat de comparaisons entre valeurs. Donnons ici tous les opérateurs de comparaison, appelés opérateurs relationnels Python.

Supposons que les variables `a` et `b` aient chacun une certaine valeur (par exemple `a = 3` et `b = 5`). Les opérateurs de comparaisons, appelés dans le jargon informatique opérateurs relationnels, possibles sont :

`a < b`	la valeur de `a` est-elle strictement inférieure à la valeur de `b`
`a > b`	la valeur de `a` est-elle strictement supérieure à la valeur de `b`
`a <= b`	la valeur de `a` est-elle inférieure ou égale à la valeur de `b`
`a >= b`	la valeur de `a` est-elle supérieure ou égale à la valeur de `b`
`a == b`	la valeur de `a` est-elle égale à la valeur de `b`
`a != b`	la valeur de `a` est-elle différente de la valeur de `b`

Attention : à ne pas confondre ou utiliser l'assignation = à la place de l'opérateur relationnel d'égalité ==.

3.2.3 Exemples de code avec des if

ANNÉE BISSEXTILE

Continuons ici d'illustrer l'utilisation du `if`, sur un exemple concret : un code qui reçoit une année et affiche si celle-ci est ou non bissextile (c'est-à-dire qui comprend 366 jours dont un 29 février).

Version simple sans elif

Dans une première approximation, nous considérons que nous avons une année bissextile tout les quatre ans et que l'année bissextile est celle dont la valeur est divisible par 4 (par exemple 2020 est bissextile).

Un nombre `x` est divisible par `4` si le reste de la division entière de `x` par `4` vaut zéro. En Python cela se teste aisément avec `x % 4 == 0`.

Le premier code complet est donc :

```
annee = int(input("Donnez l'année à tester :"))
if annee % 4 == 0:
    print("l'année", annee, "est bissextile")
else:
    print("l'année", annee, "n'est pas bissextile")
```

Exemple d'instruction if avec elif et else

En consultant l'entrée année bissextile sur wikipédia, nous voyons que la définition est plus précise.

Depuis l'ajustement du calendrier grégorien, l'année sera bissextile :
— si l'année est divisible par 4 et non divisible par 100, ou
— si l'année est divisible par 400.
Sinon, l'année n'est pas bissextile.

Observons sur l'animation Python Tutor ci-dessous que le script donne bien le bon résultat avec par exemple 2020 et 2000 qui sont bissextiles et 2100 et 2019 qui ne le sont pas (à chaque test n'oubliez pas de recharger la page sinon Python Tutor utilise la même valeur d'input). Avec l'animation, observons également le fonctionnement de l'instruction `if`, en particulier quelles sont les instructions exécutées.

Note : Animation voir cours section 3.2.3

```
annee = int(input())
if annee % 400 == 0:
    print('bissextile')
elif annee % 100 == 0:
    print('non bissextile')
elif annee % 4 == 0:
    print('bissextile')
else:
    print('non bissextile')
```

CALCUL DES RACINES D'UNE ÉQUATION DU SECOND DEGRÉ

Donnons un autre exemple plus mathématique cette fois : analysons comment écrire un code qui calcule la ou les éventuelles racines d'une équation du second degré.

Note : Ceux qui ne désirent pas faire des mathématiques maintenant peuvent simplement ignorer cet exemple qui ne donne pas de nouvelles notions Python.

Expliquons ou rappelons le problème et comment le résoudre.

Rappel : ayant l'équation du second degré

$$ax^2 + bx + c = 0$$

où a, b et c sont des valeurs réelles, pour savoir si cette équation a deux, une ou aucune racines réelles, il faut calculer le *discriminant* *delta* qui vaut :

$$delta = b^2 - 4ac$$

1. Si la valeur de $delta$ est strictement positive, l'équation a deux racines réelles :
   — $x_1 = \frac{-b-\sqrt{delta}}{2a}$
   — $x_2 = \frac{-b+\sqrt{delta}}{2a}$
2. Si la valeur de $delta$ est égale à 0, l'équation a une racine :
   — $x = \frac{-b}{2a}$
3. Si la valeur de $delta$ est strictement négative, l'équation n'a pas de racine réelle.

Le script animable suivant propose une solution au problème. Notez l'indentation des instructions qui correspondent respectivement aux parties `if`, `elif` et `else`, ainsi que l'utilisation de l'opérateur exposant un demi (par exemple : `delta ** 0.5`) pour calculer la racine carrée d'une valeur. Notez également que le code pythontutor teste d'abord le cas 3. (delta < 0) sinon le cas 2. (delta == 0) et donc le else correspond au cas 1. (delta > 0).

Animation Python Tutor de l'exemple

Testons-le avec une animation Python Tutor par exemple avec comme valeurs pour a, b, c :
— 1, -3, 2, ce qui correspond à l'équation $(x-1).(x-2) = 0$
— 1, -2, 1, ce qui correspond à l'équation $(x-1)^2 = 0$
— 1, 1, 1, ce qui correspond à l'équation $x^2 + x + 1 = 0$, où le delta est négatif (-3)
Important : rechargez la page entre chaque test pour réinitialiser l'exécution.

Note : Voir la seconde animation en section 3.2.3

Code de l'animation

```
a= float(input('a : '))
b= float(input('b : '))
c= float(input('c : '))

delta = b**2 - 4*a*c

if delta < 0:
    print(" pas de racines réelles")
elif delta == 0:
    print("une racine : ")
    print("x = ", -b/(2*a))
else:
    racine = delta**0.5
```

(suite sur la page suivante)

(suite de la page précédente)

```
    print("deux racines : ")
    print("x1 = ", (-b + racine)/(2*a))
    print("x2 = ", (-b - racine)/(2*a))
```

3.2.4 Opérateurs logiques

OPÉRATEURS LOGIQUES AND, OR ET NOT

Continuons à compléter les possibilités d'une instruction `if`. Plus précisément regardons les opérateurs qui peuvent être présents dans une condition associée à un `if` ou un `elif`.

La condition d'un `if` est une expression booléenne qui peut utiliser la valeur `True` ou `False`, ou une comparaison de valeurs avec des opérateurs relationnels, ou également les opérateurs logiques `and`, `or` et `not`.

Regardons sur quelques exemples comment fonctionnent les opérateurs logiques, appelés aussi opérateurs booléens, en supposant que la variable `x` contient une valeur entière.

- `0 <= x and x < 10` : teste si la valeur de `x` est dans l'intervalle [0 .. 10[c'est-à-dire si la valeur de `x` est supérieure ou égale à 0 et aussi strictement inférieure à 10.
- `flag = x < 0 or 10 <= x` : assigne à la variable `flag` une valeur logique vraie si la valeur de `x` est soit strictement inférieure à 0 soit supérieure ou égale à 10, et fausse sinon.
- `not (x < 0)` : est équivalent à `x >= 0`.

Notez que, pour `x` valant 6,

```
if x % 2 == 0 or  x % 3 == 0:
        print('x pair ou multiple de 3')
```

les deux tests sont vrais et donc le résultat du `or` aussi.

On parle du « ou inclusif » voulant dire que si une des deux ou si les deux parties sont vraies, le résultat est également vrai. Pour comprendre plus en détail comment l'interpréteur évalue les expressions booléennes utilisant des opérateurs logiques `and`, `or` ou `not`, il faut connaître la table de vérité de ces opérateurs.

Table de vérité du not, and et or

La table de vérité des opérateurs `and` (qui correspond au « et » en français) `or` (« ou » en français) et `not` (« non » en français) définit la sémantique de chacun des trois opérateurs logiques.

La table doit être lue comme suit : supposons que l'on ait une expression contenant des sous-expressions dont la valeur est logique (Vraie ou Fausse). Ici, les sous-expressions sont dénotées par `a` et `b`.

En fonction des différentes valeurs possibles de `a` et de `b`, Fausse (`False`) ou Vraie (`True`), la table donne la valeur de chacune des expressions `not a`, `a and b`, `a or b`.

a	b	not a	a and b	a or b
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

Associativité et priorité des opérateurs relationnels et logiques

Complétons le tableau donnant l'associativité et la priorité des opérateurs en ajoutant les opérateurs relationnels et logiques, depuis le plus prioritaire vers le moins prioritaire :

`(expression)`	
`**`	associatif à droite
`* / // %`	associatifs à gauche
`+ -`	associatifs à gauche
`< <= > >= != ==`	associatifs à gauche
`not x`	
`and`	associatif à gauche
`or`	associatif à gauche

où les opérateurs sur une même ligne ont le même niveau de priorité.

Lois de De Morgan

Les lois de De Morgan sont fréquemment utilisées pour simplifier les tests ; ayant des expressions logiques `a` et `b`
- `not (a or b)` est équivalente à `(not a) and (not b)` ;
- `not (a and b)` est équivalente à `(not a) or (not b)`.

Ainsi :

```
not(0 <= x and x < 10)
```

est équivalent à

```
(not(0 <= x)) or (not(x < 10))
```

qui par ailleurs vaut, en prenant les opérateurs duaux de <= et de < (sachant que > est l'opérateur dual de <= et vice versa) :

```
0 > x or x >= 10
```

L'EXEMPLE DE L'ANNÉE BISSEXTILE REVISITÉ

Grâce à l'utilisation des opérateurs logiques, nous pouvons maintenant produire un code plus simple du script qui affiche si une année lue en input est bissextile.

En effet, comme il n'y a que deux cas possibles (soit l'année est bissextile soit elle ne l'est pas) il est logique d'avoir une simple instruction `if` qui teste par exemple tous les cas où l'année est bissextile suivie d'un seul `else` dans le cas contraire.

Voici un exemple de code pour cela :

```
annee = int(input("Donnez l'année à tester :"))
if (annee % 4 == 0 and annee % 100 != 0) or annee % 400 == 0:
    print('bissextile')
else:
    print('non bissextile')
```

Notez que les parenthèses ont été mises pour clarifier la lecture du code, mais vu que le `and` est plus prioritaire que le `or`, elles ne sont pas nécessaires.

3.2.5 Syntaxe et sémantique de l'instruction if

SYNTAXE GÉNÉRALE DU IF

Maintenant que nous avons introduit progressivement à partir d'exemples comment fonctionne l'instruction `if`, découvrons sa forme générale. Effectivement, avant de pouvoir utiliser l'instruction `if` totalement à bon escient, il faut connaître ses syntaxe et sémantique ainsi qu'un certain nombres de règles que nous vous livrons dans cette activité.

La syntaxe générale de l'instruction `if`, telle que vous la trouverez dans les manuels de référence Python, est donnée par :

```
"if" expression ":" suite
("elif" expression ":" suite )*
["else" ":" suite]
```

où :

- `"if"`, `"elif"`, `"else"` sont des mots-clés ; les doubles apostrophes ne doivent pas être écrites,
- `expression` est une expression booléenne,
- le caractère `":"` sans les doubles apostrophes est mis après `expression` à la ligne du `if` ou des `elif` ou après le mot-clé `else`,
- `suite` est un bloc d'instructions indentées, c'est-à-dire décalées par exemple de 4 caractères vers la droite mises sur la ou les lignes suivantes,
- `( ... )*` est une méta-notation signifiant que la ligne peut ne pas être mise ou être mise une ou plusieurs fois,
- `[...]` est une méta-notation signifiant que la ligne est optionnelle, c'est-à-dire peut ou non être mise.

MOTS-CLÉS ET IDENTIFICATEURS

Dans les présentations précédentes de ce module, nous avons parlé de mots-clés. Avant de continuer, il est important d'expliquer un peu plus le but de ces mots-clés et ce à quoi il faut faire attention.

Mots-clés

Les mots-clés (keywords en anglais) Python caractérisent certains éléments du langage. Par exemple, le mot-clé `if` exprime que ce qui suit est une instruction conditionnelle. Ils doivent être écrits tel que spécifié ; par exemple pour le mot-clé `if`, les caractères doivent être des minuscules.

Python possède une trentaine de mots-clés dont la liste (pour Python 3.7) est la suivante :

False	async	del	from	lambda	return
None	await	elif	global	nonlocal	try
True	break	else	if	not	while
and	class	except	import	or	with
as	continue	finally	in	pass	yield
assert	def	for	is	raise	

Nous en avons déjà vu certains, comme `import`, `if`, `elif`, `else`, `and`, `or`, ...

Nous en verrons d'autres dans ce cours, mais pas tous. Certains correspondent en effet à des concepts avancés.

Identificateurs

En dehors des mots-clés, un programme Python utilise entre autres des variables et des fonctions. Chaque variable et fonction porte généralement un nom. Par exemple dans les exemples précédents nous avions `x`, `a` et `delta` comme noms de variables ou `int`, `input`, `print` comme noms de fonctions prédéfinies ou de type.

Dans le jargon informatique, ces noms de variables, fonctions, types sont appelés identificateurs. Comme son nom l'indique, un identificateur est le nom qui identifie quelque chose. Un identificateur n'est pas un mot-clé.

Comme les mots-clés sont des mots réservés pour dénoter quelque chose de précis dans le langage, il ne peuvent évidemment pas être utilisés à d'autres fins. Ainsi, essayer d'appeler une variable avec le nom d'un mot-clé va générer une erreur lors de l'exécution. Par exemple si nous demandons à l'interpréteur d'exécuter

```
  if = 5
SyntaxError: invalid syntax
```

il m'indique qu'il y a une erreur de syntaxe puisque l'interpréteur s'attend, après le mot-clé `if`, à avoir une expression booléenne et une suite bien précise et pas un symbole d'assignation comme ici.

Lors de la rédaction d'un code Python, nous devons donc bien faire attention à ne pas utiliser un mot-clé pour nommer par exemple une variable.

Notons que réutiliser, comme nom d'identificateur, le nom d'un identificateur prédéfini est possible même si c'est une très mauvaise idée. Ainsi on peut écrire :

```
print = 3
```

mais par la suite

```
print("Résultat =", res)
```

donnera une erreur puisque `print` ne correspond plus à la fonction prédéfinie, mais à une variable entière et donc lui accoler des parenthèses ne veut plus rien dire.

Ne faites donc pas cela !

N'OUBLIEZ PAS L'INDENTATION

Respectez parfaitement l"indentation des instructions Python, c'est-à-dire le fait d'écrire les lignes de code associées à un `if`, `elif` ou `else`, en les décalant de par exemple 4 caractères.

Faites-y toujours bien attention sachant que :

```
x = int(input('donnez une valeur entière : '))
if x > 0 :
   print('la valeur est strictement positive')
print('Au revoir')
```

où, quelle que soit la valeur lue, le message "Au revoir" est affiché à l'écran, est différent du code :

```
x = int(input('donnez une valeur entière : '))
if x > 0 :
   print('la valeur est strictement positive')
   print('Au revoir')
```

où le message "Au revoir" n'est affiché que si la valeur lue est strictement positive.

L'oubli ou une mauvaise indentation est une erreur très fréquente quand vous débutez en programmation Python. Cela peut générer des erreurs de syntaxe ou pire, des erreurs dans les instructions qu'il fallait exécuter pour que le code soit correct.

IF IMBRIQUÉS

Un code Python peut également avoir des `if` imbriqués par exemple pour distinguer des sous-cas. Il faut bien faire attention à l'indentation comme le montrent les deux bouts de code suivants :

Code 1

```
if a > 0 :
   if  b > 0 :
       print("cas 1")
   else :
       print("cas 2")
```

où le `else` correspond au `if` imbriqué (celui le plus à l'intérieur) et

Code 2

```
if a > 0 :
   if  b > 0 :
       print("cas 1")
else :
   print("cas 2")
```

où le `else` correspond au `if` global.

Pour comprendre finement ce que fait ce code avec différentes valeurs des variables `a` et `b`, n'hésitez pas à tester ces deux exemples dans Python Tutor.

Pour cela, vous pouvez exécuter étape par étape les deux codes des animations, générées par Python Tutor (comme habituellement avec Python Tutor, utilisez les boutons « back » et « forward » et communiquez les données dans la fenêtre de soumission puis cliquez sur « submit ») :

Animation Python Tutor du code 1

Note : Voir la première animation de la section 3.2.5 du cours en ligne

---

Animation Python Tutor du code 2

Note : Voir la seconde animation de la section 3.2.5 du cours en ligne

---

ÉVALUATION PARESSEUSE

Il nous reste à voir une dernière règle qui pourra s'avérer très utile sur l'évaluation des expressions booléennes. Python est paresseux quand il évalue certaines expressions.

En clair, en Python,

- si l'expression booléenne est un `and` et la valeur de gauche est évaluée à Faux, la partie à droite du `and` n'est jamais évaluée et la réponse au test complet est Faux ;
- de façon duale, si l'expression booléenne est un `or` et si la valeur de gauche est Vraie, la partie à droite du `or` n'est jamais évaluée et la réponse au test complet est Vrai.

L'évaluation paresseuse peut être utile si par exemple la première partie de la condition détermine si le second test peut être réalisé.

Par exemple : avec deux variables `a` et `b` contenant chacune une valeur entière positive, le code suivant est correct :

```
if  a != 0 and b % a == 0:
   print(b, "est un multiple de", a)
```

Il teste d'abord la condition `a != 0` ; si ce test est faux, l'évaluation de la condition complète s'arrête et a la valeur `False`. Si par contre la condition `a != 0` a la valeur `True`, l'évaluation de la condition `b % a == 0` est effectuée par l'interpréteur.

Cette façon de faire permet donc de s'assurer que le calcul du modulo `(b % a)` ne va pas donner une erreur à l'exécution (division par zéro) et sinon de ne pas faire cette opération.

3.2.6 Quiz sur l'instruction if

MINI QUIZ SUR CE QUE L'ON A DEJA VU

Note : Voir le quiz de la section 3.2.6

3.3 Pratique de l'instruction if

3.3.1 Mettons ensemble en pratique ce que nous venons de voir

UN PETIT JEU DE DEVINETTE

Maintenant que vous maîtrisez les subtilités de l'instruction `if`, passons à la pratique !

Écrivons d'abord un programme qui propose un petit jeu à l'utilisateur. Par la suite nous compléterons ce jeu et nous reviendrons à ce programme plus tard dans ce module afin d'illustrer d'autres instructions.

Commençons par une première version simplifiée dans laquelle :

Consignes : le programme :
- choisit aléatoirement un nombre entre 0 et 5 sans en afficher la valeur (et donc sans que l'utilisateur connaisse cette valeur) et le place dans la variable `secret` ;
- demande à l'utilisateur de deviner la valeur choisie ;
- affiche `"gagné !"` si l'utilisateur trouve la bonne réponse et
- affiche `"perdu ! La valeur était "` suivi de la valeur de `secret` dans le cas contraire.

Petite astuce pour vous permettre de vous lancer

En informatique, la génération de nombres aléatoires se fait généralement par une fonction qui fait des calculs en fonction de paramètres divers et produit un résultat dans l'intervalle demandé. On parle donc plutôt de génération de nombre pseudo aléatoire. En effet, si le nombre est vu par l'utilisateur comme étant aléatoire, il provient en réalité de calculs précis effectués par l'ordinateur.

En Python, le module `random` peut être utilisé à cette fin. En particulier `random` contient la fonction prédéfinie `randint(a, b)` où `a` et `b` sont des valeurs entières (par exemple 0 et 5). À chaque nouvel appel, `randint` génère un nombre pseudo aléatoire dans l'intervalle entre la valeur `a` et la valeur `b` toutes deux comprises (`[a, b]`).

Le code peut donc commencer par :

```
import random
secret = random.randint(0, 5)
```

À vous de jouer ! Écrivez un script dans votre IDE (Thonny ou PyCharm par exemple) qui résout l'exercice proposé avec les consignes données ci-dessus.

3.3.2 Jeu de devinette : proposition de solution

PROPOSITION DE SOLUTION

Quand plusieurs personnes écrivent du code pour résoudre un problème, comme ici avec le petit jeu de devinette, les codes sont très certainement différents même s'ils peuvent tous parfaitement résoudre le problème. Malgré cela, il peut quand même être intéressant de comparer les solutions pour voir laquelle semble la plus claire ou la plus efficace. Ainsi, si vous voulez comparer votre solution avec un autre code qui résout le problème quand vous aurez terminé, ou si vous avez besoin d'un petit coup de pouce, la vidéo suivante propose une solution.

PROPOSITION DE SOLUTION

Note : Voir la vidéo de la section 3.3.2 : Proposition de solution

CONTENU DE LA VIDÉO

La vidéo précédente présente une solution du petit jeu de devinette. Le code d'une solution est donné plus bas.

CODE D'UNE SOLUTION

Ci-dessous la solution proposée dans la vidéo de la section 3.3.2.

```
""" Petit jeu de devinette (version 1)
Auteur: Thierry Massart
Date : 5 avril 2018
"""

import random
secret = random.randint(0,5)
choix_utilisateur = int(input("Donnez votre choix pour la valeur secrète : "))
if secret == choix_utilisateur:
   print("gagné !")
else:
   print("perdu ! La valeur était", secret)
print("Au revoir !")
```

3.3.3 Le if : mise en pratique autonome : exercice UpyLaB 3.1

MISE EN PRATIQUE AUTONOME

Nous voici arrivés à la troisième phase d'apprentissage en ce qui concerne l'instruction `if`. Dans l'activité qui va suivre, nous vous proposons d'utiliser notre exerciseur UpyLaB pour tester de manière autonome plusieurs exercices qui demandent d'utiliser l'instruction `if`.

Vous pouvez désormais réaliser les exercices UpyLaB 3.1 à 3.8 du Module 3, donnés aux pages qui suivent. Avant cela, lisez bien les rappels et recommandations ci-dessous.

RAPPELS IMPORTANTS

Note : Nous n'insisterons jamais assez sur le fait que, lors du développement de chacun de vos codes, il est plus que vivement conseillé d'utiliser un environnement de développement de type PyCharm ou Python Tutor pour *tester* ou même *déboguer* pas par pas votre code avant de passer aux tests *UpyLaB*. En effet, faire tester directement par UpyLaB un code dont vous n'avez pas une certaine confiance dans le fait qu'il soit correct s'avère souvent être un gouffre au niveau du temps étant donné qu'UpyLaB est moins précis qu'un environnement de développement pour vous expliciter où se situent les éventuels problèmes qui ne manqueront pas d'apparaître.

La démarche la plus efficace pour développer du code en Python est donc :

1. de comprendre exactement ce qui vous est demandé,
2. de développer une solution dans votre IDE en la testant, éventuellement avec Python Tutor, pour vous assurer qu'elle résout correctement le problème posé,

3. de tester l'ensemble de votre code dans UpyLaB (en le transférant dans UpyLaB grâce au copier / coller du code depuis votre IDE) pour faire valider votre script.

Attention : Rappelez-vous également de bien suivre les autres **Conseils et consignes lors de la réalisation d'exercices UpyLaB** (cf section 2.5) et que le but de réaliser les exercices UpyLaB est de vous rendre autonome en programmation. C'est vous qui devez trouver la ou une solution, en terme de code Python, à chaque exercice demandé. Le forum ne doit donc pas être l'endroit où des codes complets ou substantiels sont échangés. Vos éventuelles questions doivent être précises et les réponses ponctuelles.

N'oubliez pas ce qui a été dit en début de cours sur les exercices UpyLaB !

1) Contrairement aux quiz, les exercices UpyLaB peuvent être testés autant de fois que vous le désirez.
2) Pour enregistrer votre réponse dans un exercice UpyLaB vous devez cliquer sur le bouton « Vérifier » même si vous savez que votre code n'est pas correct ; ce n'est pas un souci puisque le nombre de fois que vous vérifiez chaque exercice UpyLaB n'est pas limité.
3) Attention, si pour un exercice UpyLaB, la dernière vérification est négative, la plateforme ne retient pas si une vérification précédente avait été positive. En clair, la plateforme ne vous accorde les points que si la dernière vérification de cet exercice est positive.

EXERCICE UPYLAB 3.1 (Vert)

Énoncé

Écrire un programme qui lit 3 nombres entiers, et qui, si au moins deux d'entre eux ont la même valeur, imprime cette valeur (le programme n'imprime rien dans le cas contraire).

Exemple 1

Avec les données lues suivantes :

```
2
1
2
```

le résultat à imprimer vaudra :

```
2
```

Exemple 2

Avec les données lues suivantes :

```
1
2
3
```

le programme n'affichera rien.

Exemple 3

Avec les données lues suivantes :

```
42
42
42
```

le résultat à imprimer vaudra :

```
42
```

Consignes

- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
- En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.1*.

3.3.4 Exercice UpyLaB 3.2 (Vert)

Énoncé

Écrire un programme qui, si `temperature` (entier lu sur input correspondant à la température maximale prévue pour aujourd'hui) est strictement supérieur à 0, teste si `temperature` est inférieur ou égal à 10, auquel cas il imprime le texte :

- `Il va faire frais.`

et qui, si `temperature` n'est pas supérieur à 0, imprime le texte :

- `Il va faire froid.`

Dans les autres cas, le programme n'imprime rien.

Exemple 1

Avec la donnée lue suivante :

```
1
```

le résultat à imprimer vaudra :

```
Il va faire frais.
```

Exemple 2

Avec la donnée lue suivante :

```
20
```

le programme n'affichera rien.

Exemple 3

Avec la donnée lue suivante :

```
-1
```

le résultat à imprimer vaudra :

```
Il va faire froid.
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
— Faites attention d'écrire les messages à l'identique de ce qui est demandé (majuscule au début de la ligne, une espace entre chaque mot, etc) . Un conseil pour avoir des messages identiques est de les copier depuis l'énoncé pour les coller dans votre code.
— Lors de l'affichage des résultats, en cas d'erreur dans certains tests, UpyLaB pourra marquer : « Le résultat attendu était : aucun résultat ». Cela voudra bien dire qu'il ne faut rien imprimer dans ce cas.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.2*.

3.3.5 Exercice UpyLaB 3.3 (Vert)

Énoncé

Écrire un programme qui lit trois entiers a, b et c en input. Ensuite :
- si l'entier c est égal à 1, alors le programme affiche la valeur de $a + b$;
- si c vaut 2, alors le programme affiche la valeur de $a - b$;
- si c est égal à 3, alors l'output sera la valeur de $a.b$ (produit de a par b);
- enfin, si la valeur 4 est assignée à la variable c, alors le programme affiche la valeur de $a^2 + a.b$;
- et si c contient une autre valeur, le programme affiche le message `Erreur`.

Exemple 1

Avec les données lues suivantes :

```
3
2
1
```

le résultat à imprimer vaudra :

```
5
```

Exemple 2

Avec les données lues suivantes :

```
3
2
4
```

le résultat à imprimer vaudra :

```
15
```

Exemple 3

Avec les données lues suivantes :

```
3
2
5
```

le résultat à imprimer vaudra :

```
Erreur
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu en respectant l'orthographe, les majuscules / minuscules, les espacements, etc.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.3*.

3.3.6 Exercice UpyLaB 3.4 (Vert)

Énoncé

Écrire un programme qui teste la parité d'un nombre entier lu sur input et imprime `True` si le nombre est pair, `False` dans le cas contraire.

Exemple 1

Avec la donnée lue suivante :

```
13
```

le résultat à imprimer vaudra :

```
False
```

Exemple 2

Avec la donnée lue suivante :

```
42
```

le résultat à imprimer vaudra :

```
True
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Pour tester la parité d'un nombre, on peut s'intéresser au reste de la division de ce nombre par 2 et utiliser l'opérateur modulo.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.4*.

3.3.7 Exercice UpyLaB 3.5 (Vert)

Énoncé

Écrire un programme qui lit en entrée deux nombres entiers strictement positifs, et qui vérifie qu'aucun des deux n'est un diviseur de l'autre.

Si tel est bien le cas, le programme imprime `True`. Sinon, il imprime `False`.

Exemple 1

Avec les données lues suivantes :

```
6
42
```

le résultat à imprimer vaudra :

```
False
```

Exemple 2

Avec les données lues suivantes :

```
5
42
```

le résultat à imprimer vaudra :

```
True
```

Consignes

— Notez qu'il n'est pas demandé de vérifier que les deux nombres en entrée sont strictement positifs. Vous pouvez supposer que ce sera le cas pour toutes les entrées proposées par UpyLaB lors des tests.
— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Dire qu'un nombre entier x est un diviseur d'un nombre entier y revient à dire que le reste de la division euclidienne de y par x est égal à 0.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.5*.

3.3.8 Exercice UpyLaB 3.6 (Vert)

Énoncé

Écrire un programme qui imprime la moyenne géométrique $\sqrt{a.b}$ (la racine carrée du produit de a par b) de deux nombres positifs a et b de type float lus en entrée.

Si au moins un de ces nombres est strictement négatif, le programme imprime le texte « Erreur ».

Exemple 1

Avec les données lues suivantes :

```
1.0
2.0
```

le résultat à imprimer vaudra approximativement :

```
1.4142135623730951
```

Exemple 2

Avec les données lues suivantes :

```
-1.0
2.0
```

le résultat à imprimer vaudra :

```
Erreur
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`float(input())` et non `float(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Pour calculer la racine carrée d'un nombre positif, on pourra trouver une fonction utile dans le module math, ou se rappeler que cela correspond à la puissance d'exposant $\frac{1}{2}$.
— Si vous calculez la puissance d'exposant $\frac{1}{2}$, pensez bien aux priorités des différents opérateurs.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) ou dans Python Tutor avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.6*.

3.3.9 Exercice UpyLaB 3.7 (Vert)

Énoncé

Dans mon casino, ma roulette comporte 13 numéros de 0 à 12 comme montrés ci-dessous :

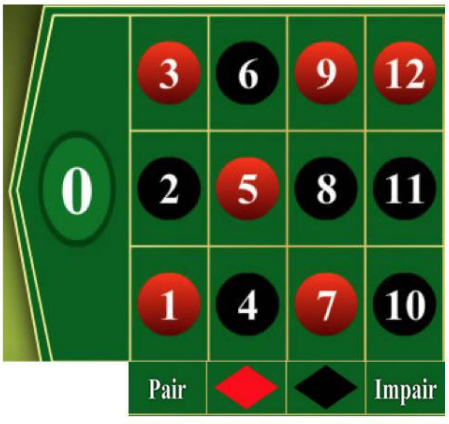


Fig. 3.1 – Roulette

Le joueur a plusieurs types de paris possibles :
— il peut choisir de parier sur le numéro sortant, et dans ce cas, s'il gagne, il remporte douze fois sa mise ;
— il peut choisir de parier sur la parité du numéro sortant (pair ou impair), et dans ce cas, s'il gagne, il remporte deux fois sa mise ;
— enfin, il peut choisir de parier sur la couleur du numéro sortant (rouge ou noir), et dans ce cas aussi, s'il gagne, il remporte deux fois sa mise.

Si le joueur perd son pari, il ne récupère pas sa mise.

Pour simplifier, on suppose que le numéro 0 n'est ni rouge ni noir, mais est pair. Pour simplifier encore, on suppose que le joueur mise systématiquement 10 euros.

Écrire un programme qui aide le croupier à déterminer la somme que le casino doit donner au joueur.

Le programme lira, dans l'ordre, deux nombres entiers en entrée : le pari du joueur (représenté par un nombre entre 0 et 16, voir description plus bas), et le numéro issu du tirage (nombre entre 0 et 12). Le programme affichera alors le montant gagné par le joueur.

Entrées pour le pari du joueur :

- nombre entre 0 et 12 : le joueur parie sur le numéro correspondant
- 13 : le joueur parie sur pair
- 14 : le joueur parie sur impair
- 15 : le joueur parie sur la couleur rouge
- 16 : le joueur parie sur la couleur noire.

Exemple 1

Avec les données lues suivantes :

```
7
9
```

qui indiquent que le joueur parie sur le numéro 7 et que le numéro sorti est le 9,

le résultat à imprimer vaudra donc

```
0
```

Exemple 2

Avec les données lues suivantes :

```
16
4
```

qui indiquent que le joueur parie sur la couleur noire et que le numéro sorti est le 4 (qui est noir),

le résultat à imprimer vaudra donc

```
20
```

soit deux fois sa mise de 10 euros.

Exemple 3

Avec les données lues suivantes :

```
7
7
```

qui indiquent que le joueur parie sur le numéro 7 et que le numéro sorti est bien le 7,

le résultat à imprimer vaudra donc

```
120
```

soit douze fois sa mise de 10 euros.

Consignes

— Ne mettez pas d'argument dans les `input` : `data = input()` et non `data = input("Donnée suivante :")` par exemple.
— Le résultat doit juste faire l'objet d'un `print(res)` sans texte supplémentaire (pas de `print("résultat = ", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Lisez bien l'énoncé. Que représente chacune des deux entrées ? Avez-vous parfaitement analysé les différents cas ?
— Même si le hasard intervient dans cette situation, ici il ne faut pas utiliser le module random. Nous ne simulons pas le tirage de la roulette, c'est l'utilisateur (le croupier) qui saisira les deux entrées correspondant au pari du joueur, et au résultat du tirage.
— Pensez à utiliser les opérateurs logiques, mais faites attention à leur priorité : voir section 3.2.3.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.7*.

3.3.10 Exercice UpyLaB 3.8 (Rouge)

Énoncé

Les cinq polyèdres réguliers de Platon sont représentés ci-dessous, avec la formule de leur volume.

Source des images de polyèdres : Vikidia, l'encyclopédie pour les jeunes, qui explique aux enfants et à ceux qui veulent une présentation simple d'un sujet (https://fr.vikidia.org/wiki/Polyèdre).

Nom	Volume	Image
Tétraèdre	$\frac{\sqrt{2}}{12}a^3$	
Cube	a^3	
Octaèdre	$\frac{\sqrt{2}}{3}a^3$	
Dodécaèdre	$\frac{15+7\sqrt{5}}{4}a^3$	
Icosaèdre	$\frac{5(3+\sqrt{5})}{12}a^3$	

Écrire un programme qui lit :

— la première lettre en majuscule du nom du polyèdre (`"T"`, `"C"`, `"O"`, `"D"` ou `"I"`),

— la longueur de l'arête du polyèdre,

et qui imprime le volume du polyèdre correspondant.

Si la lettre lue ne fait pas partie des cinq initiales, le programme imprime le message `"Polyèdre non connu"`.

Exemple 1

Avec les données lues suivantes :

```
C
2.0
```

le résultat à imprimer vaudra :

```
8.0
```

Exemple 2

Avec les données lues suivantes :

```
I
2.0
```

le résultat à imprimer vaudra approximativement :

```
17.4535599249993
```

Exemple 3

Avec les données lues suivantes :

```
A
2.0
```

le résultat à imprimer vaudra :

```
Polyèdre non connu
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`float(input())` et non `float(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— La fonction `input` retourne systématiquement un résultat de type chaîne de caractères. Si ce qui est à lire est un texte ou un simple caractère, on peut donc utiliser directement le retour de cette fonction, mais s'il s'agit d'un nombre, il faut penser à le convertir à l'aide des fonctions `int` ou `float`.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.8*.

3.4 Les instructions répétitives while et for

3.4.1 L'instruction while

INTRODUISONS L'INSTRUCTION WHILE

La présente section vous explique tout ce qui vous sera nécessaire sur l'instruction `while`. Cette vidéo introduit l'instruction `while` qui permet d'exécuter certaines instructions de façon répétitive tant qu'une certaine condition est vérifiée. Comme le traitement peut s'exécuter plusieurs fois, nous parlerons pour l'instruction `while` mais aussi pour l'instruction `for` que nous verrons juste après, d'instructions répétitives `while` et `for`. Encore une fois, nous partons d'un exemple simple pour expliquer le fonctionnement de l'instruction `while`, et ensuite nous élaborons pour montrer différentes possibilités.

Notons aussi que dans le jargon informatique, les instructions répétitives `while` et `for` sont également appelées « boucles » `while` ou `for` pour marquer cette notion de répétition.

INTRODUCTION DE L'INSTRUCTION RÉPÉTITIVE WHILE

Note : Voir la vidéo de la section 3.4.1 : L'instruction while

---

CONTENU DE LA VIDÉO

La vidéo précédente présente l'instruction répétitive `while`.

Script réalisé dans la vidéo

```
DISTANCE = 3844.0e5
nombre_pliages = 0
epaisseur = 0.0001
while epaisseur < DISTANCE :
   epaisseur = 2 * epaisseur
   nombre_pliages = nombre_pliages + 1
print('nombre de pliages nécessaire : ', nombre_pliages)
```

La vidéo en bref

La vidéo introduit l'instruction répétitive `while` avec un petit exemple.

La syntaxe d'une instruction while est

```
while condition :
    instructions
```

avec le mot-clé `while`, suivi d'une expression booléenne que l'on appelle souvent condition de continuation, suivie de « : », puis de toutes les instructions qui sont indentées par rapport à cette première ligne et qui constituent les instructions associées au while.

Dans le jargon informatique, on appelle ces instructions le «corps de la boucle while».

Lors de l'exécution d'une instruction while, la condition de continuation est évaluée, et si elle est vraie, le corps de la boucle while est exécuté, ensuite, on recommence cette séquence : évaluation de la condition, exécution du corps de la boucle jusqu'au moment où la condition est évaluée à faux.

Notons que cette condition peut être directement fausse ; dans ce cas, le corps de la boucle n'est pas exécuté et l'interpréteur passe directement à l'instruction après l'instruction while.

C'EST FACILE D'ALLER SUR LA LUNE

42 correspond à la première valeur de x telle que $0.0001.2^x \geq 3.844.10^8$. Notons pour les forts en math que cela correspond à la première solution entière de l'inéquation $384400000 < 0.0001.2^n$ c'est-à-dire à la première valeur entière supérieure à $log_2(3844000000000)$ (en python `ceil(log(3840000000000, 2))` où `log` et `ceil` (valeur plafond) ont été importées du module `math` (`from math import ceil, log`). C'est donc très facile d'aller sur la Lune : il suffit de plier une feuille 42 fois et de se mettre dessus !

3.4.2 Syntaxe du while et calcul du plus grand commun diviseur

SYNTAXE DE L'INSTRUCTION RÉPÉTITIVE WHILE

La syntaxe de l'instruction répétitive `while` est :

```
while condition:
   instructions
```

où `condition` est une expression booléenne appelée condition de continuation et `instructions` est une instruction ou une séquence d'instructions indentées par rapport à la ligne `while condition:`, que l'on nomme généralement corps de la boucle `while`.

CALCUL DU PLUS GRAND COMMUN DIVISEUR

Un autre exemple plus mathématique mais simple, qui illustre parfaitement l'utilisation de l'instruction répétitive `while` est le calcul du plus grand commun diviseur de deux nombres entiers positifs. Rappelons d'abord ce qu'est le plus grand commun diviseur.

Définition du plus grand commun diviseur (pgcd)

Ayant deux nombres entiers positifs x et y (par exemple 132 et 36),

le *plus grand diviseur (pgcd)* des deux nombres est le nombre d qui est :

- **un diviseur entier à la fois de x et de y, c'est-à-dire que**
  - x divisé par d donne un nombre entier,
  - et de même pour y divisé par d,
- **et de plus tel qu'il n'existe pas d'autre entier strictement plus grand que** d également diviseur entier à la fois de x et de y.

Ainsi on peut vérifier que 12 est le plus grand commun diviseur de 132 et 36.

Calcul du plus grand commun diviseur (pgcd) avec la méthode d'Euclide améliorée

Le calcul du plus grand commun diviseur est un très bel exemple d'utilisation de l'instruction `while` vu sa simplicité. La mini capsule vidéo suivante explique le problème du calcul du `pgcd(x,y)` et sa solution simple en Python.

MÉTHODE DE CALCUL DU PGCD(X,Y)

Note : Voir la vidéo de la section 3.4.2 : Calcul du PGCD

CONTENU DE LA VIDÉO

La vidéo précédente présente un script qui calcule le plus grand commun diviseur de deux nombres lus en entrée.

Script réalisé dans la vidéo

```
x = int(input("x = "))
y = int(input("y = "))
while y > 0:
    x, y = y, x % y
    print(x, y)
print("pgcd = ", x)
```

La vidéo en bref

Le script met directement en application deux propriétés mathématiques du pgcd, soit :

— Si `x` est positif et `y` strictement positif : `pgcd(x, y)` est équivalent à `pgcd(y, x % y)`
Par exemple `132 % 36` vaut `24` et donc `pgcd(132, 36)` vaut `pgcd(36, 24)`
ce qui est intéressant car `y` et `x % y` seront plus petits que `x` et `y`.

— `pgcd(x, 0)` vaut `x`.

FONCTION GCD DU MODULE MATH

Notons que Python met à votre disposition la fonction `gcd` du module `math`, qui calcule le plus grand commun diviseur de deux nombres entiers.

3.4.3 Conjecture de Syracuse

CONJECTURE DE SYRACUSE

Mettons en pratique ce que nous venons de voir sur l'instruction `while`.

Aidons les mathématiciens à valider une conjecture pourtant simple qui leur donne du fil à retordre. Le texte sur la conjecture qui suit est repris de l'article Wikipedia sur la conjecture de Syracuse.

D'abord par un peu d'histoire :

En 1952, lors d'une visite à Hambourg, le mathématicien allemand Lothar Collatz expliqua le « problème 3x+1 » à Helmut Hasse. Ce dernier le diffusa en Amérique à l'Université de Syracuse sous le nom de « suite de Collatz » ou de « suite de Syracuse ». Entre temps, le mathématicien polonais Stanislas Ulam le répand dans le Laboratoire national de Los Alamos. Dans les années 1960, le problème est repris par le mathématicien Shizuo Kakutani qui le diffuse dans les universités Yale et Chicago.

Définissons le problème :

La suite de Collatz pour un nombre entier n strictement positif est définie comme suit :

— la première valeur de la suite est le nombre n lui-même,
— si n est pair la valeur suivante sera n divisé par 2,
— sinon la valeur suivante est 3n + 1 (n multiplié par 3 auquel on rajoute 1).

En répétant l'opération, on obtient une suite d'entiers positifs dont chacun ne dépend que de son prédécesseur.

Par exemple, à partir de 14, on construit la suite des nombres : 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2…

C'est ce qu'on appelle la suite de Collatz ou suite de Syracuse du nombre 14.

Après que le nombre 1 a été atteint, la suite des valeurs (1,4,2,1,4,2…) se répète indéfiniment en un cycle de longueur 3, appelé cycle trivial.

Si l'on était parti d'un autre entier, en lui appliquant les mêmes règles, on aurait obtenu une suite de nombres différente. A priori, il serait possible que la suite de Syracuse de certaines valeurs de départ n'atteigne jamais la valeur 1, soit qu'elle aboutisse à un cycle différent du cycle trivial, soit qu'elle diverge vers l'infini. Or, on n'a jamais trouvé d'exemple de suite obtenue suivant les règles données qui n'aboutisse pas à 1 et, par suite, au cycle trivial.

La conjecture de Syracuse est l'hypothèse mathématique selon laquelle la suite de Syracuse de n'importe quel entier strictement positif atteint 1.

Cette conjecture mobilisa tant les mathématiciens durant les années 1960, en pleine guerre froide, qu'une plaisanterie courut selon laquelle ce problème faisait partie d'un complot soviétique visant à ralentir la recherche américaine.

Un grand mathématicien du 20ème siècle, Paul Erdős a dit, à propos de la conjecture de Syracuse : « Les mathématiques ne sont pas encore prêtes pour de tels problèmes ».

IMPLÉMENTATION DE LA CONJECTURE DE SYRACUSE

Nous voulons écrire un code qui génère la suite de Syracuse d'un nombre `n` lu sur input, jusqu'à ce qu'elle atteigne la valeur 1.

Développons le problème.

Le nombre `n` est lu sur input :

```
n = int(input('valeur du nombre n dont on veut tester la conjecture'))
```

Calculer la valeur suivante peut se faire avec une instruction `if`

```
if n % 2 == 0 :  # si un nombre entier modulo 2 vaut 0, il est pair
   n = n // 2
else:                  # cas où le nombre est impair
   n = 3 * n + 1
```

Enfin pour tester si la suite satisfait la conjecture, il faut continuer à calculer de nouvelles valeurs jusqu'à ce que `n` vaille 1 :

```
while n != 1:
   if n % 2 == 0 :  # si un nombre entier modulo 2 vaut 0, il est pair
       n = n // 2
   else:                  # cas où le nombre est impair
       n = 3 * n + 1
```

Notons que si la conjecture n'est pas satisfaite, ce programme boucle indéfiniment, ce qui signifie que ici que la boucle `while` ne s'arrêtera jamais sauf si l'utilisateur coupe l'exécution du programme. Des informaticiens ont testé la conjecture pour tous les entiers jusqu'à des valeurs de l'ordre de 2^{60} sans trouver de contre-exemple. Mais à l'heure actuelle aucune preuve mathématique n'existe qu'il n'en existe pas un plus grand qui ne la satisfait pas.

N'hésitez pas à mettre tous les morceaux de code ensemble pour obtenir un programme complet. Pour voir si votre code complet ressemble au nôtre, une solution vous est proposée ici.

Une solution pour Syracuse

```
n = int(input('entier strictement positif : '))
while n != 1:
   print(n)
   if n % 2 == 0:
       n = n // 2
   else:
       n = n * 3 + 1
print(n)  # imprime 1
```

3.4.4 Deux canevas classiques rencontrés avec une boucle while

TRAITEMENT JUSQU'À CE QU'UNE CERTAINE CONDITION SOIT VÉRIFIÉE

Assez souvent le programmeur qui a un code à produire retrouve des « canevas » classiques qui lui permettent rapidement de savoir quelles instructions utiliser. Le canevas type pour une instruction `while` contient souvent les quatre parties `initialisation`, `condition`, `traitement`, `changement` :

```
initialisation
while condition:
    traitement
    changement
```

avec :

- une partie `initialisation` avant le `while` permettant après de tester la `condition de continuation`;
- la ligne contenant le `while` avec la `condition de continuation` qui détermine si une itération supplémentaire doit être réalisée dans le while ;
- la partie `traitement` dans le corps du `while`;
- une partie que l'on peut qualifier de `changement` à la fin du corps du `while` qui constitue une sorte de *réinitialisation* qui va permettre le test suivant de la `condition de continuation`.

Par exemple, dans le code *Syracuse* que nous venons de présenter :

Les parties `initialisation` et `changement` initialisent et modifient `n` qui sera testé au niveau de la `condition` de while ; la partie `traitement` imprime la nouvelle valeur de `n`.

```
n = "première valeur"
while "n ne correspond pas à la valeur pour arrêter":
    traitement
    n = "nouvelle valeur"
```

LECTURE D'UNE SUITE DE DONNÉES

Il est fréquent que l'on lise une suite de données et fasse un certain traitement sur ces données tant que le code ne lit pas une valeur qui spécifie la fin de la suite des données à lire.

Par exemple si l'on doit lire une suite de données jusqu'à une valeur dite « sentinelle » valant `"F"`, le canevas classique, avec deux lignes d'input nécessaires, est le suivant :

```
x = input('Première donnée : ')
while x != 'F':
    traitement
    x = input('Donnée suivante : ')
```

N'oubliez pas que la fonction `input()` renvoie un résultat de type chaîne de caractères (par exemple `'1'`, `'32'`, `'-66'` ou `'F'`) et que la fonction `int(x)` ou `float(x)` transforme une chaîne de caractères `x`, qui dénote respectivement un nombre entier, et fractionnaire, en la valeur correspondante de type `int` ou `float`. Vous pouvez bien sûr utiliser les deux fonctions en séquence directe, par exemple :

```
n = int(input())
```

ou non, par exemple :

```
x = input()
if x != 'F':
    n = int(x)
    ...
...
```

Certains des exercices UpyLaB qui suivent utiliseront ces canevas.

3.4.5 Des programmes qui bouclent indéfiniment

PROGRAMME QUI BOUCLE

Une des hantises du programmeur est un programme qui *boucle*, c'est-à-dire qui réexécute sans fin et de façon erronée la même séquence d'instructions. L'utilisation de l'instruction `while` est une des causes possibles d'un tel programme qui boucle indéfiniment, comme illustré dans la capsule vidéo suivante :

EXEMPLES DE PROGRAMMES QUI BOUCLENT INDÉFINIMENT

Note : Voir la vidéo de la section 3.4.5 : Programmes qui bouclent

CONTENU DE LA VIDÉO

La vidéo précédente présente des exemples d'instructions répétitives `while` qui bouclent indéfiniment.

Scripts réalisés dans la vidéo

```
i = 0
while i >= 0 :
    i = i + 1
    print(i)
```

```
i = 0
while i >= 0 :
    i = i + 1
print("le programme n'arrivera jamais ici")
```

La vidéo en bref

Des codes peuvent ne pas s'arrêter tout seuls, comme montré avec les scripts donnés dans la vidéo, où des instructions `while` ont la condition de continuation qui n'est jamais évaluée à faux durant leur exécution. Dans ce cas, il faudra arrêter l'exécution de ces codes généralement erronés, grâce à une touche clavier ou un bouton spécifique d'interruption d'exécution.

3.4.6 L'instruction for

INSTRUCTION FOR

La seconde instruction répétitive de Python est l'instruction `for`, également appelée boucle `for`.

La vidéo suivante illustre le fonctionnement de l'instruction `for` par un exemple très simple. Cet exemple introduit également la notion de séquence sur laquelle nous reviendrons plus en détails dans le module de cours suivant. Nous allons voir que l'instruction `for` répète un traitement pour tous les éléments d'une séquence.

INTRODUCTION DE L'INSTRUCTION RÉPÉTITIVE FOR

Note : Voir la vidéo de la section 3.4.6 : L'instruction for

CONTENU DE LA VIDÉO

La vidéo précédente présente l'instruction répétitive `for`.

Scripts réalisés dans la vidéo

```
for c in "Bonjour":
    print(c)
```

```
for i in range(5):
    print(i)
```

```
somme = 0
for val in range(5) :
     somme = somme + val
print(somme)
```

La vidéo en bref

L'instruction for réalise un traitement donné pour tous les éléments d'une séquence.

La vidéo montre trois scripts : un premier qui réalise un traitement sur tous les caractères d'une chaîne de caractères, et deux autres qui utilisent la fonction `range` pour faire un traitement avec une séquence de valeurs entières.

La syntaxe de base d'une instruction `for` est ensuite donnée dans la vidéo (voir plus loin dans le cours).

RANGE

Dans la vidéo précédente nous avons vu que le code :

```
for i in range(5):
     print(i)
```

affiche

```
0
1
2
3
4
```

Notez donc que le code affiche 5 valeurs de 0 à 4 (donc 5 non compris).

Nous verrons en section 3.5.11 que `range` peut également être utilisé avec 2 ou 3 arguments ce qui étend ses possibilités.

3.4.7 Syntaxe du for, carrés, étoiles et autres polygones réguliers

SYNTAXE DE L'INSTRUCTION FOR

Nous avons vu dans la vidéo précédente que la syntaxe de l'instruction répétitive `for` est :

```
for c in sequence:
   instructions
```

où :

— `c` est la *variable de contrôle* qui reçoit séquentiellement chacune des valeurs de la séquence
— `sequence` est, ou génère, la séquence de valeurs utilisée
— `instructions` constitue le *corps de la boucle* `for`

Pour que votre code fonctionne correctement, il faut absolument que les `instructions` dans le *corps du for* ne modifient pas la variable `c` (et ce même si Python ne l'interdit pas explicitement). Si vous modifiez `c`, votre code va souvent produire des effets assez aléatoires !

DESSIN D'UN CARRÉ, D'UN POLYGONE RÉGULIER OU D'UNE ÉTOILE AVEC TURTLE

Pour mettre en pratique l'instruction `for`, utilisons à nouveau le module `turtle` pour dessiner des polygones.

Une boucle `for` permet de dessiner très simplement un carré (par exemple dont les côtés ont une longueur de 100).

```
import turtle
for i in range(4):
   turtle.forward(100)
   turtle.left(90)
turtle.done()
```

À vous ! Reprenez ce code pour écrire et exécuter dans votre IDE (Thonny ou PyCharm par exemple) un script qui trace un polygone régulier à n côtés avec n au moins égal à 3, ou une étoile à n branches avec n au moins égal à 5 et de valeur impaire.

Rappelons-nous (module 2 Activité 2.6.3) que pour dessiner un polygone à n côtés (par exemple $n = 5$), l'angle intérieur entre deux côtés est de $360°/n$. Si n vaut 5 cela donne donc $72°$.

Rappelons-nous également que si l'on veut dessiner des étoiles à n branches et en supposant n impair au moins égal à 5, l'angle intérieur sera de $(n - 1) * 180°/n$ (par exemple $144°$ pour n valant 5).

POUR LES AMOUREUX DES ÉTOILES OU DES MATHÉMATIQUES

Dans le paragraphe précédent nous avons expliqué comment dessiner une étoile ayant un nombre impair, au moins égal à 5, de branches. En supposant que l'on désire dessiner une étoile à n branches par une succession de segments contigus, une telle étoile peut être dessinée pour toutes les valeurs de n impaires à partir de 5 et la plupart des valeurs paires.

En dessous de 5 nous ne pouvons tracer d'étoile. Une exception notoire au dessus de 5 est l'étoile à 6 branches qui, comme le montre la figure « Etoile à 6 branches », demande d'effectuer deux tracés distincts où à chaque fois, on dessine un triangle.

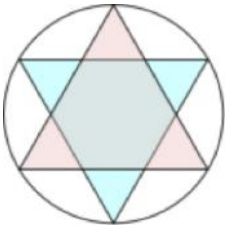


Fig. 3.2 – Étoile à 6 branches

Confectionnons à présent un code qui reçoit en entrée le nombre n de branches et qui, s'il est possible de tracer une étoile à n branches en un seul tracé contigu, la dessine avec turtle.

Pour trouver comment faire, analysons la situation avec une valeur n paire. Prenons l'exemple de n valant 8. Le principe est de prendre un cercle et de le découper en n sections comme on découpe une tarte en n parts égales. Le dessin ci-dessous représente une telle découpe pour n valant 8.

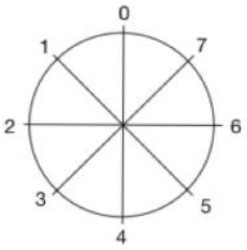


Fig. 3.3 – Découpe du cercle en 8 parts égales.

Ensuite, le principe est de partir du point 0 et de tracer un segment vers un autre point. Nous répétons cela n fois. Partant de 0, le point suivant peut par exemple être le point 3. Ensuite, l'idée est de continuer avec le même incrément, donc le point 6 et ainsi de suite, en ajoutant 3 à chaque fois. Quand on arrive à un nombre plus grand que 7, on effectue un modulo 8. Dans notre exemple, après 6 on aura 6 + 3 donne 9 ; dans ce cas 9 modulo 8 nous dit que le point suivant est 1.

Donc le tracé passe séquentiellement par les points 0, 3, 6, 1, 4, 7, 2, 5, et retour à 0, comme le montre le schéma suivant.

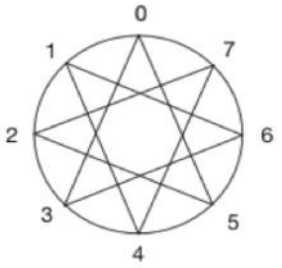


Fig. 3.4 – Étoile à 8 branches

De façon générale avec n sommets et n au moins égal à 5 (en dessous on ne pourra tracer une étoile), partant du point 0, il faut trouver un incrément inc qui modulo n fera passer par tous les points avant de revenir au point 0. L'incrément inc devra être plus grand que 1 sinon, nous traçons un polygone et non pas une étoile. Nous prenons aussi inc inférieur à n/2 sinon le tracé se fera en traçant par la droite et non par la gauche. Il se peut que plusieurs incréments soient possibles. Si nous désirons l'étoile avec les branches les plus fines possibles, il faut que inc soit le plus proche de n/2. Quand n est impair la valeur inc telle que `n = 2 . inc + 1` fonctionne toujours (en Python il suffit si n est impair de définir `inc = n // 2`). Si n est pair ce n'est pas aussi facile. Mais les mathématiques nous disent qu'il faut que inc soit la plus grande valeur entière strictement plus petite à `n / 2` telle que le `pgcd(n, inc)` soit égal à 1 .

Le code Python suivant effectue le travail :

```
""" trace une étoile à n côtés, si elle peut l'être sans lever la plume"""


import turtle
from math import gcd # fonction du module math qui calcule le pgcd de 2 nombres

LONGUEUR = 100 # taille de chaque segment de l'étoile

n = int(input("Combien de branches désirez-vous ? :"))
```

(suite sur la page suivante)

(suite de la page précédente)

```
inc = (n-1) // 2
while gcd(n, inc) > 1:
   inc = inc - 1
if inc == 1 :
   print("Impossible de dessiner une étoile à", n, "branches en un tenant")
else:
   angle =  180 - (n - 2 * inc) * 180 / n
   for i in range(n):
       turtle.forward(LONGUEUR)
       turtle.left(angle)
   turtle.done()
```

Notons qu'à partir de 5 branches, l'étoile à 6 branches est la seule qui ne puisse être tracée avec ce programme. En effet 6 est la seule valeur n entière positive telle qu'aucune valeur entière positive i dans l'intervalle entre 1 non compris et n/2 non compris, soit telle que `pgcd(i, n) == 1`.

Bons dessins d'étoiles !

3.4.8 Quiz sur while et for

QUIZ POUR RÉCAPITULER SUR LES INSTRUCTIONS WHILE ET FOR

Note : Voir le quiz de la section 3.4.8

3.5 Code avec while et for dans la pratique

Mettons ensemble en pratique ce que nous avons vu sur les boucles `while` et `for` Python. Avant de vous proposer de résoudre, de façon autonome, des exercices UpyLaB, commençons par des exercices avec correction pour comparer votre solution à la nôtre et vous guider dans votre apprentissage pratique : une activité sur la suite de Fibonacci suivie d'une activité de dessin d'un pavé hexagonal.

3.5.1 La suite de Fibonacci

SUITE DE FIBONACCI

Selon le site « image des mathématiques » du CNRS :

La suite de Fibonacci commence ainsi :

0 1 1 2 3 5 8 13 21 34 55 89 144 233 . . .

Ses deux premiers termes sont 0 et 1, et ensuite, chaque terme successif est la somme des deux termes précédents. Ainsi

- 0 + 1 = 1
- 1 + 1 = 2
- 1 + 2 = 3
- 2 + 3 = 5
- 3 + 5 = 8
- etc.

Son inventeur est Léonard de Pise (1175 - v.1250), aussi connu sous le nom de Leonardo Fibonacci. Introduite comme problème récréatif dans son fameux ouvrage Liber Abaci, la suite de Fibonacci peut être considérée comme le tout premier modèle mathématique en dynamique des populations ! En effet, elle y décrit la croissance d'une population de lapins sous des hypothèses très

simplifiées, à savoir : chaque couple de lapins, dès son troisième mois d'existence, engendre chaque mois un nouveau couple de lapins, et ce indéfiniment.

Partons donc d'un couple de lapins le premier mois. Le deuxième mois, on n'a toujours que ce même couple, mais le troisième mois on a déjà 2 couples, puis 3 couples le quatrième mois, 5 couples le cinquième mois, etc. La croissance de cette population est bel et bien décrite par la suite de Fibonacci en partant de 0 et 1 comme deux premiers nombres de cette suite.

Nous vous proposons d'écrire deux petits programmes qui calculent et impriment les premiers termes de la suite de Fibonacci dans deux cas de figure :

— **problème 1** : pour les n premiers termes
— **problème 2** : pour tous les termes inférieurs à une valeur n donnée.

Pour simplifier, supposons que pour les 2 problèmes, n est supérieur ou égal à 2.

Ces deux problèmes illustrent parfaitement la règle suivante qui guide le choix, lorsque l'on doit répéter un traitement, de l'utilisation de l'instruction `for` ou `while` :

Avertissement : Règles : Lorsqu'un traitement répétitif doit être réalisé par un programme Python,

— **l'utilisation d'une instruction répétitive `for` est à favoriser**
quand nous savons facilement, au début de l'exécution de la boucle, combien d'itérations devront être réalisées ou si l'on demande de traiter tous les éléments d'une séquence comme nous le verrons au module sur le séquences ;
— **l'utilisation d'une instruction répétitive `while` est à favoriser**
quand a priori le nombre d'itérations n'est pas facilement déterminable avant le début de son exécution.
— N'essayez pas cette expérience avec votre couple de lapins !

Les deux codes commencent donc par « lire une valeur entière n » (c'est-à-dire, lire une valeur entière et assigner cette valeur à la variable `n`).

Comme on ne sait pas calculer n termes en une simple séquence d'assignations et de `if` puisque que n n'est connu qu'à l'exécution quand le script du code est déjà écrit, il est clair que pour les deux codes, il faut utiliser une instruction répétitive (une boucle). Une façon habituelle de coder dans ce type d'exemple est de calculer un nouveau terme à chaque itération.

Encore faut-il savoir comment mettre cela en musique et si c'est une instruction `while` ou `for` qu'il faut utiliser.

Pour les deux problèmes qui nous préoccupent, comme pour le calcul du plus grand commun diviseur, une façon classique de calculer un terme suivant est d'avoir deux variables, que l'on nomme par exemple `prec` et `succ`, qui vont contenir l'avant-dernier et le dernier terme calculés.

Initialement on a

```
prec = 0
succ = 1
```

À chaque itération, il faut calculer le terme suivant mais aussi conserver l'avant-dernier, c'est-à-dire celui qui était dans `succ` pour nous permettre de continuer.

Typiquement on pourra écrire :

```
prec, succ = succ, prec + succ
```

SUITE DE FIBONACCI : PROBLÈME 1

Si l'on veut écrire les n premiers termes de la suite, il suffit de les calculer un à un et de les afficher, par exemple avec

```
for i in range(n):
    prec, succ = succ, prec + succ
    print(succ)
```

Plus précisément, le code complet animé par Python Tutor du problème 1 est :

Code pour le problème 1

```
n = int(input('nombre de termes à calculer de la suite de Fibonacci : '))
prec = 0
succ = 1
print(prec, end = ' ')
print(succ, end = ' ')
for i in range(n-2):
   prec, succ = succ, prec + succ
   print(succ, end = ' ')
print()
```

Commentaires sur le code

Le `range(n-2)` est dû au fait que les 2 premiers termes ont déjà été imprimés et qu'il en reste donc `n-2` à calculer.

Notons aussi l'explicitation de l'argument nommé `end` dans les `print`, pour imprimer les résultats sur la même ligne (avec `end = ' '`, chaque print est séparé par une espace), et le dernier `print()` permet de passer à la ligne après avoir imprimé la dernière valeur.

SUITE DE FIBONACCI : PROBLÈME 2

Pour le problème 2, on ne sait pas a priori combien d'itérations devront être effectuées. Dans ce cas, un `while` est la bonne instruction à utiliser.

Le code animé par Python Tutor du problème 2 est donc :

Code pour le problème 2

```
n = int(input('borne supérieur à ne pas dépasser pour calculer la suite de Fibonacci : '))
prec = 0
succ = 1
print(prec, end = ' ')
while succ < n :
   print(succ, end = ' ')
   prec, succ = succ, prec + succ
print()
```

Commentaires sur le code

Notons qu'ici en plus d'utiliser un `while`, `succ` est imprimé juste après le test (condition de continuation du `while`) pour s'assurer qu'il faut effectivement l'imprimer.

Le programmeur devra toujours essayer d'avoir non seulement un code correct pour toutes les exécutions possibles, mais également le plus efficace possible tout en étant le plus clair possible. Nous reviendrons sur cet aspect plus tard.

Note : une instruction `for` peut en général facilement être traduite par un `while` même si cela donne du code plus compliqué et donc moins lisible ; c'est donc à déconseiller. Par exemple :

AUTRE EXEMPLE DE CODE QUI UTILISE UNE INSTRUCTION FOR

Une instruction for peut être réécrite en un while comme le montre l'exemple ci-dessous :

```
for i in range(10) :
     print(i)
```

peut être écrit comme suit :

```
i = 0
while i < 10 :
    print(i)
    i = i + 1
```

Note : Codes animés avec Python Tutor : voir cours en ligne en section 3.5.1

Pour être précis, les deux codes n'ont pas un effet totalement équivalent : à la sortie du `for`, `i` vaut 9 et pas 10 comme à la sortie du `while`.

3.5.2 Un pavé hexagonal avec des for

CODE D'UN PAVÉ HEXAGONAL AVEC DES FOR IMBRIQUÉS

Un exemple beaucoup plus difficile à ce stade de votre apprentissage est d'écrire un code qui utilise `turtle` pour faire un pavé hexagonal tel que demandé au module précédent, mais en ayant le code le plus succinct possible grâce à l'utilisation d'instructions `for`.

Fig. 3.5 – Pavé coloré

En effet, on peut constater qu'un tel pavé est constitué de trois losanges, chacun constitué de quatre côtés de même taille, mais reliés avec des angles alternativement de 60° ou de 120°.

Comme nous n'avons pas encore vu la notion de séquence qui nous aidera pour faire l'exemple complet, essayez d'écrire le code qui utilise en particulier des instructions `for` pour tracer l'ensemble des lignes de l'hexagone, mais sans changer de couleur ni remplir les surfaces, tel que montré ici :

Aide : on peut voir que le code suivant imprime quatre fois la valeur d'une variable angle en alternant entre les deux valeurs 120 et 60 (120, 60, 120, 60). En effet ce code utilise le fait que 180 - 120 vaut 60 et 180 - 60 vaut 120.

```
angle = 120
for j in range(4):
       print(angle)
       angle = 180 - angle
```

Par ailleurs,

— le premier losange commence son tracé par un segment de ligne horizontal,

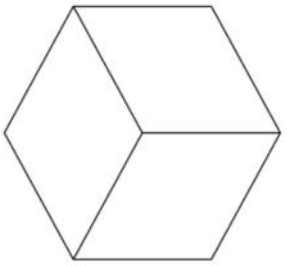


Fig. 3.6 – Pavé non coloré

— le second, par un segment qui forme un angle de 120° par rapport au premier segment,
— le troisième, par un segment qui forme un angle de 120° par rapport au second segment.

En résumé, il faut répéter trois fois :

— tracer un losange (où à chaque fois il faut tracer quatre segments)
— ensuite tourner à droite de 120°.

Le canevas général sera donc :

```
for i in range(3) : # à chaque itération, trace un losange
    for j in range(4) : # à chaque itération, trace un segment
        ...
```

À vous de jouer pour écrire un programme complet réalisant le tracé du pavé hexagonal.

PROPOSITION DE SOLUTION

Vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir une autre solution ? Nous vous en proposons une :

```
""" auteur: Thierry Massart
   date : 9 avril 2018
   but du programme : trace avec turtle les contours d'un pavé hexagonal
"""
import turtle
for i in range(3):  # à chaque itération, trace un losange
   angle = 120
   for j in range(4): # à chaque itération, trace un segment
       turtle.forward(100)
       turtle.left(angle)
       angle = 180 - angle
   turtle.right(120)
turtle.hideturtle()
turtle.done()
```

3.5.3 Mise en pratique autonome : exercices UpyLaB 3.9 et suivants

MISE EN PRATIQUE AUTONOME AVEC UPYLAB

Maintenant que nous avons ensemble rédigé des scripts avec des « boucles », il vous reste l'ultime étape dans l'apprentissage de ce module : devenir autonome dans la rédaction de tels codes. Nous vous demandons de réaliser les exercices UpyLaB 3.9 à 3.17 du Module 3 proposés dans les pages qui suivent. En particulier, l'exercice UpyLaB 3.14 est une version plus élaborée du petit jeu de devinette présenté en section 3.3 du présent module.

EXERCICE UPYLAB 3.9 (Vert)

Énoncé

Écrire un programme qui demande à l'utilisateur combien de plis de papier sont nécessaires pour se rendre sur la Lune, et pose la question tant que l'utilisateur n'a pas saisi la bonne réponse. Si la réponse saisie par l'utilisateur n'est pas correcte, le programme affiche le message `"Mauvaise réponse."`, puis pose à nouveau la question. Si la réponse saisie par l'utilisateur est correcte, le programme affiche le message `"Bravo !"`, et s'arrête. Exemple

Dans cet exemple d'exécution, le texte est affiché par le programme, alors que les nombres sont saisis par l'utilisateur :

```
Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : 666
Mauvaise réponse.
Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : 3
Mauvaise réponse.
Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : 42
Bravo !
```

Consignes

— UpyLaB va vérifier le bon fonctionnement du programme en comparant ce qui est affiché à l'écran avec ce qu'il attend. Veillez donc à bien respecter le texte à afficher (casse, espaces, ponctuation. . .).
— En particulier, pour lire les données utilisez précisément l'instruction :
`int(input("Combien de plis sont-ils nécessaires pour se rendre sur la Lune ? : "))`
et pour afficher utilisez précisément les instructions
`print("Mauvaise réponse.")`
et
`print("Bravo !")`
— Pour rappel, la réponse attendue a été calculée par le script de Sébastien dans la vidéo présentant la boucle `while`, en section 4.3.1. Cette réponse n'est pas à faire calculer par votre programme, qui se contente de la demander à l'utilisateur.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une solution à l'exercice : vidéo d'une solution de l'exercice UpyLaB 3.9

Conseils

— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui

fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée. * Votre code n'est pas accepté par UpyLaB et pourtant semble afficher ce qui est attendu ? Veillez à bien respecter la casse et les espaces, et n'hésitez pas à faire un copier/coller de la phrase à afficher depuis l'énoncé.

3.5.4 Exercice UpyLaB 3.10 (Vert)

Énoncé

Écrire un programme qui calcule la taille moyenne (en nombre de salariés) des Petites et Moyennes Entreprises de la région.

Les tailles seront données en entrée, chacune sur sa propre ligne, et la fin des données sera signalée par la valeur sentinelle -1. Cette valeur n'est pas à comptabiliser pour le calcul de la moyenne, mais indique que l'ensemble des valeurs a été donné.

Après l'entrée de cette valeur sentinelle -1, le programme affiche la valeur de la moyenne arithmétique calculée.

On suppose que la suite des tailles contient toujours au moins un élément avant la valeur sentinelle -1, et que toutes ces valeurs sont positives ou nulles.

Exemple 1

Avec les données lues suivantes :

```
11
8
14
5
-1
```

le résultat à imprimer vaudra :

```
9.5
```

Exemple 2

Avec les données lues suivantes :

```
12
6
7
-1
```

le résultat à imprimer vaudra approximativement :

```
8.333333333333334
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il sera utile de créer deux variables, une qui stockera la somme des valeurs entrées et l'autre leur nombre, et que l'on actualisera après chaque lecture.
— N'oubliez pas de supprimer les textes à l'intérieur des appels à `input` lorsque vous soumettez le code à UpyLaB.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.10*.

3.5.5 Note sur la fonction print

PRINT

Nous avons vu que la fonction `print()` affiche l'ensemble des valeurs données en argument. Ainsi le code suivant :

```
date_1 = 1515
date_2 = 1789
print("Bataille de Marignan", date_1)
print("Révolution française", date_2)
```

affiche les deux lignes

```
Bataille de Marignan 1515
Révolution française 1789
```

Notez donc que :

1) chaque print se termine par un passage à la ligne suivante ;
2) une espace sépare chaque valeur affichée.

Il se peut que l'on ne désire pas passer à la ligne après un appel à `print`, ou que l'on ne veuille pas avoir une espace comme séparateur entre les valeurs.

Il est possible de modifier ces deux comportements de la fonction `print`, en explicitant ce que l'interpréteur doit utiliser comme chaîne de caractères de séparation (argument nommé `sep`) et de fin d'affichage (argument nommé `end`).

Par exemple :

```
date_1 = 1515
date_2 = 1789
print("Bataille de Marignan", date_1, sep=' : ', end='/')
print("Révolution française", date_2, sep=' : ', end='\n-----\n')
```

affiche les deux lignes

```
Bataille de Marignan : 1515/Révolution française : 1789
-----
```

3.5.6 Exercice UpyLaB 3.11 (Vert)

Énoncé

Écrire un programme qui lit sur input une valeur naturelle `n` et qui affiche à l'écran un carré de `n` caractères X (majuscule) de côté.

Exemple 1

Avec la donnée lue suivante :

```
6
```

le résultat à imprimer vaudra :

```
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
XX
XX
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple).
— Il n'est pas demandé de tester si la valeur *n* est bien positive ou nulle, vous pouvez supposer que ce sera toujours le cas pour les valeurs transmises par UpyLaB.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Si votre programme est rejeté par UpyLaB alors qu'il semble produire le résultat attendu, vérifiez bien que vous n'ajoutez pas d'espaces superflus, en début ou en fin de ligne, et que votre code n'affiche pas une ligne vide supplémentaire.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.11*.

3.5.7 Exercice UpyLaB 3.12 (Bleu)

Énoncé

Cet exercice propose une variante de l'exercice précédent sur le carré de X.

Écrire un programme qui lit sur input une valeur naturelle `n` et qui affiche à l'écran un triangle supérieur droit formé de X (voir exemples plus bas).

Exemple 1

Avec la donnée lue suivante :

```
6
```

le résultat à imprimer vaudra :

```
XXXXXX
 XXXXX
  XXXX
   XXX
    XX
     X
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
XX
 X
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple).
— Il n'est pas demandé de tester si la valeur *n* est bien positive ou nulle, vous pouvez supposer que ce sera toujours le cas pour les valeurs transmises par UpyLaB.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— La concaténation (qui consiste à coller deux textes ensemble) et l'opérateur `*` sur les chaînes de caractères pourront s'avérer utiles ici.
— Si votre programme est rejeté par UpyLaB alors qu'il semble produire le résultat attendu, vérifiez bien que vous n'ajoutez pas d'espaces superflus, en début ou en fin de ligne, et que votre code n'affiche pas une ligne vide supplémentaire.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.12*.

3.5.8 Exercice UpyLaB 3.13 (Bleu)

Énoncé

Écrire un programme qui additionne des valeurs naturelles lues sur input et affiche le résultat.

La première donnée lue ne fait pas partie des valeurs à sommer. Elle détermine si la liste contient un nombre déterminé à l'avance de valeurs à lire ou non :

— si cette valeur est un nombre positif ou nul, elle donne le nombre de valeurs à lire et à sommer ;
— si elle est égale à -1, cela signifie qu'elle est suivie d'une liste de données à lire qui sera terminée par le caractère `"F"` signifiant que la liste est terminée.

Exemple 1

Avec les données lues suivantes :

```
4
1
3
5
7
```

qui indiquent qu'il y a 4 données à sommer : `1 + 3 + 5 + 7`,

le résultat à imprimer vaudra donc

```
16
```

Exemple 2

Avec les données lues suivantes :

```
-1
1
3
5
7
21
F
```

qui indiquent qu'il faut sommer : `1 + 3 + 5 + 7 + 21`,

le résultat à imprimer vaudra donc

```
37
```

Exemple 3

Avec la donnée :

```
0
```

qui indique qu'il faut sommer `0` nombre,

le résultat à imprimer vaudra donc

```
0
```

Consignes

- Ne mettez pas d'argument dans l'"`input` :
- `data = input()` et pas
- `data = input("Donnée suivante :")` par exemple ;
- Le résultat doit juste faire l'objet d'un `print(res)` sans texte supplémentaire (pas de `print("résultat =", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Dans la cas où la liste est terminée par le caractère `"F"`, envisagez de lire l'input, puis de tester si la valeur lue est différente du caractère `"F"`, avant de chercher à la convertir en int et à l'ajouter à ce qui a déjà été sommé ;
- Utilisez un `for` dans le cas où le nombre de valeurs à sommer est connu, un `while` dans le cas contraire.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 3.13*.

3.5.9 Exercice UpyLaB 3.14 (Rouge)

Énoncé

Dans cet exercice, nous revenons sur le petit jeu de devinette.

Écrire un programme qui génère de manière (pseudo) aléatoire un entier (nombre secret) compris entre 0 et 100. Ensuite, le joueur doit deviner ce nombre en utilisant le moins d'essais possible.

À chaque tour, le joueur est invité à proposer un nombre et le programme doit donner une réponse parmi les suivantes :

- « Trop grand » : si le nombre secret est plus petit que la proposition et qu'on n'est pas au maximum d'essais
- « Trop petit » : si le nombre secret est plus grand que la proposition et qu'on n'est pas au maximum d'essais
- « Gagné en *n* essai(s) ! » : si le nombre secret est trouvé
- « Perdu ! Le secret était *nombre* » : si le joueur a utilisé **six essais** sans trouver le nombre secret.

Exemple 1

Une partie gagnante (après la génération du nombre à deviner) :

NB : Les nombres sont les valeurs saisies par l'utilisateur, et les textes sont imprimés par le programme.

```
50
Trop grand
8
Trop petit
20
Trop petit
27
Gagné en 4 essai(s) !
```

Exemple 2

Une partie gagnante (après la génération du nombre à deviner) :

```
50
Trop grand
24
Trop petit
37
Trop petit
43
Trop grand
40
Trop petit
41
Perdu ! Le secret était 42
```

Consignes

— Attention, au dernier essai, le programme ne doit afficher ni « Trop petit » ni « Trop grand », mais le verdict comme illustré plus haut.
— Pour qu'Upylab puisse tester que votre solution est correcte, il faut que vous respectiez strictement la séquence décrite dans l'énoncé. Si par exemple, vous n'affichez pas « Trop petit » ou « Trop grand », le nombre suivant ne sera pas fourni par le système et votre solution sera considérée comme incorrecte.
— En pratique, pour la génération du nombre secret, vous devez débuter votre code comme suit :

```
import random
NB_ESSAIS_MAX = 6
secret = random.randint(0, 100)
```

et ne pas faire d'autre appel à `randint` ou à une autre fonction du module `random`.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une solution à l'exercice : vidéo d'une solution de l'exercice UpyLaB 3.18 (à partir de 14'47 »)

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Pour tester notre code, UpyLaB va exécuter le programme en ajoutant l'instruction `random.seed(argument)`, où `argument` est une certaine valeur. Cette instruction permet de générer les mêmes suites de nombres aléatoires lors des appels aux fonctions du module `random` à chaque exécution du programme.
Si, avec votre IDE, vous souhaitez reproduire le comportement exact d'UpyLaB lors des tests, il faudra donc rajouter cette instruction juste après l'import du module `random`, en remplaçant `argument` par la valeur indiquée dans le test d'UpyLaB. Vous serez ainsi sûr que votre code génèrera le même nombre secret qu'UpyLaB, ce qui peut faciliter le débogage en cas de test invalidé par exemple.
Notez bien que l'argument de la fonction `seed` n'est pas le nombre à deviner, mais un paramètre qui permet de le générer.
Attention, cette instruction ne doit pas figurer dans le code que vous soumettez à UpyLaB.
— Si vous rencontrez l'erreur `EOF Error`, vérifiez que votre programme se termine bien dès que l'utilisateur découvre le nombre secret et ce, même en moins de 6 essais, ou bien dès que le nombre maximal d'essais est atteint.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.14*.

3.5.10 Exercice UpyLaB 3.15 (Rouge)

Énoncé

Comme mes écureuils s'ennuyaient, je leur ai fabriqué une roue avec des barreaux pour qu'ils puissent faire de l'exercice.

La roue fait 100 barreaux. Pour m'y retrouver, je les numérote de 0 à 99.

Très vite je me suis rendu compte que chacun utilisait la roue en faisant des sauts identiques (Up fait des sauts de 7 barreaux à chaque fois, Py des sauts 9, LaB des sauts de 13, ...).

Je mets une noisette sur un des barreaux de la roue. Aidez-moi à savoir si un de mes écureuils va l'attraper sachant que je vais mettre l'écureuil qui fait le test, par exemple Up, sur le barreau 0 et que je connais son comportement (Up fait toujours des sauts de 7 barreaux) et que je sais le numéro de barreau, différent de 0, où se trouve la noisette (Up n'aura donc pas la noisette au départ).

Écrire un programme qui teste si pour une configuration donnée, un écureuil va ou non atteindre un moment la noisette. Il reçoit deux valeurs entières en entrée, une valeur `saut` et une valeur `position_cible` toutes deux entre 1 et 99.

Le programme va calculer une valeur `position_courante`, initialement la valeur 0, et vérifier si en calculant de façon répétitive la valeur `position_courante`, celle-ci aboutira un moment à la valeur `position_cible`.

Notez que pour calculer la valeur suivante de `position_courante` (initialement mise à 0), il faut incrémenter la valeur actuelle de `position_courante` de la valeur `saut` et ensuite, si le résultat est plus grand ou égal à 100, calculer la position en faisant un modulo 100 de la valeur obtenue (ce qui donne à chaque fois une valeur `position_courante` entre 0 et 99). (Notez que l'on peut systématiquement faire le modulo 100 du résultat sans tester si `position_courante` est ou non supérieur à 100 pour obtenir sa bonne valeur).

Notez également, pour ne pas épuiser mon écureuil sans fin, que s'il atteint à nouveau le barreau `0`, j'arrête l'expérience sachant qu'il prendra toujours les mêmes barreaux sans jamais atteindre `position_cible` (la noisette).

À la fin votre programme dira si oui ou non la noisette a été atteinte ou non.

En pratique, après avoir lu les deux valeurs `saut` et `position_cible`, votre programme affichera chaque valeur de `position_courante` sur une ligne différente à partir de la seconde valeur (pas la `position_courante` initiale qui vaut toujours 0). La dernière `position_courante` affichée sera soit 0 soit la dernière valeur de `position_courante` avant qu'elle n'aie la valeur de `position_cible`, si l'écureuil trouve la noisette. Votre programme terminera en affichant, sur une nouvelle ligne, le message donnant le résultat :

— `"Cible atteinte"` si l'écureuil a trouvé la noisette,
— `"Pas trouvée"` si l'écureuil est revenu en position 0 sans trouver la noisette.

Vous pouvez supposer que les valeurs lues sont bien des entiers qui respectent les consignes.

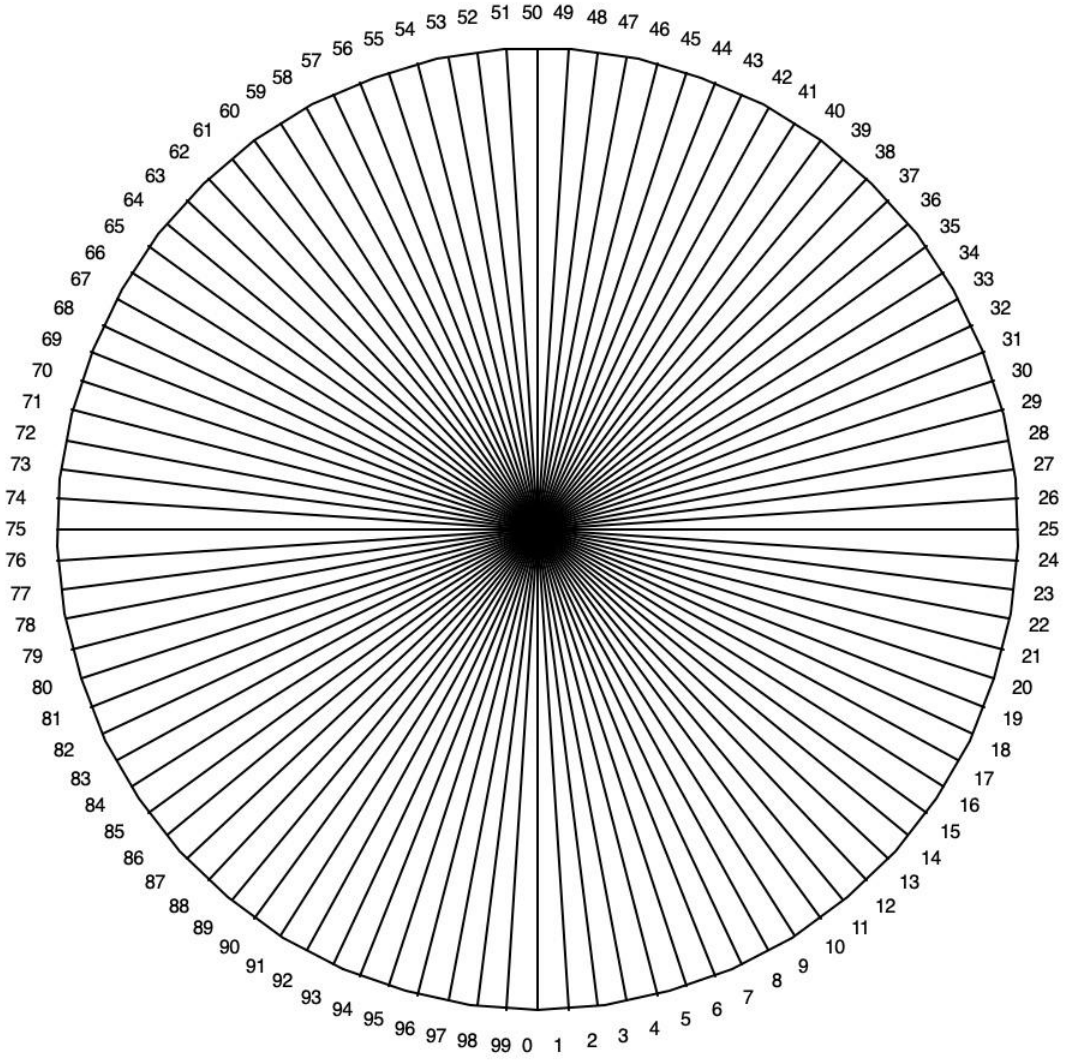


Fig. 3.7 – Roue de mes écureuils

Exemple 1

Avec les données lues suivantes :

```
9
7
```

le résultat à imprimer vaudra :

```
9
18
27
36
45
54
63
72
81
90
99
8
17
26
35
44
53
62
71
80
89
98
Cible atteinte
```

Exemple 2

Avec les données lues suivantes :

```
8
7
```

le résultat à imprimer vaudra :

```
8
16
24
32
40
48
56
64
72
80
88
96
4
12
20
28
```

(suite sur la page suivante)

(suite de la page précédente)

```
36
44
52
60
68
76
84
92
0
Pas trouvée
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : ")` par exemple), et à afficher précisément le texte demandé ; par exemple : `print("Cible atteinte")` et non `print("La cible a été atteinte")` par exemple.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il sera utile de définir trois variables, une qui stockera la valeur `saut` (inchangée tout au long du programme puisqu'elle est utilisée comme incrément), une qui stockera la valeur `position_cible` où se trouve la noisette, et enfin une qui stockera la valeur `position_courante`. **Nous vous conseillons** au début d'assigner à cette variable `position_courante` la valeur `saut` soit la position le l'écureuil après un saut (la position initiale de l'écureuil valant 0).
— Nous rappelons que l'opérateur modulo (% en Python) donne le reste de la division euclidienne. Ainsi, 42 % 5 est égal à 2 car 42 = 5 * 8 + 2.
— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 3.15*.

3.5.11 range(debut, fin, pas)

RANGE

Nous avons déjà utilisé `range` lors de la présentation de l'instruction for à la section précédente. Expliquons plus complètement les différentes formes d'utilisation de cette instruction.

`range` génère une séquence de valeurs entières. On l'utilise en lui donnant de 1 à 3 arguments entiers.

range avec un argument

Nous avons vu l'utilisation avec un seul argument, comme avec :

```
for i in range(5):
    print(i)
```

l'instruction `range` affiche

```
0
1
2
3
4
```

soit une séquence avec les valeurs entières depuis 0 compris jusque 5 non compris.

range avec deux arguments

Si nous utilisons dans un code `range` avec deux arguments, le premier argument spécifie la première valeur de la séquence (si la séquence n'est pas vide), et le deuxième argument la borne finale (non incluse). Par exemple :

```
somme = 0
for i in range(1,11):
    somme = somme + i
    print(somme)
```

affiche

```
1
3
6
10
15
21
28
36
45
55
```

soit les différentes valeurs de `somme` sommant en 10 étapes les valeurs entières de 1 à 10.

range avec trois arguments

Si nous utilisons dans un code `range` avec trois arguments, les deux premiers arguments spécifient la première valeur de la séquence et la borne finale non comprise. Le troisième argument donne le *pas* ou *incrément* pour calculer la valeur suivante à partir de la valeur précédente. Par exemple :

```
for i in range(0, 51, 5):
    print(i, end = ' ')
print()
```

affiche les différentes valeurs de i soit :

```
0 5 10 15 20 25 30 35 40 45 50
```

depuis la valeur 0 jusqu'à la valeur 51 non comprise par pas de 5 (à chaque étape, pour avoir la valeur suivante, on incrémente la précédente de 5).

Dans cet exemple les multiples de 5, entre 0 et 50 compris, sont affichés sur la même ligne et séparés d'une espace grâce à l'argument nommé `end = ' '` ; et après la dernière valeur affichée, le `print()` placé après l'instruction `for`, permet de passer à la ligne suivante pour tout affichage ultérieur.

Notons que :

— la valeur du *pas* peut être négative (par exemple : `range(10, -1, -1)`)
— `range` peut donner une séquence vide (par exemple : `range(0)`, `range(1, 1)`, `range(0, 10, -1)`)

3.5.12 Exercice UpyLaB 3.16 (Rouge)

Énoncé

Ecrivez un code qui lit un nombre entier strictement positif n et affiche sur n lignes une table de multiplication de taille $n \times n$, avec, pour *i* entre 1 et n, les *n* premières valeurs multiples de *i* strictement positives sur la *ième* ligne. Ainsi, les n premiers multiples de 1 strictement positifs (0 non compris) sont affichés sur la première ligne, les n premiers multiples de 2 sur la deuxième, et caetera.

Exemple 1

Avec la valeur lue suivante :

```
3
```

le résultat à afficher sera :

```
1   2   3
2   4   6
3   6   9
```

Exemple 2

Avec la valeur lue suivante :

```
10
```

le résultat à afficher sera :

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Exemple 3

Avec la valeur lue suivante :

```
1
```

le résultat à afficher sera :

```
1
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé.
— Il n'est pas requis que les nombres soient alignés verticalement. Pour cet exercice, UpyLaB ne tiendra pas compte du nombre d'espaces séparant les nombres sur chaque ligne, ni de la présence d'espace en fin de ligne.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— L'utilisation de `range(...)` peut être utile.

3.5.13 Exercice UpyLaB 3.17 (Rouge)

Énoncé

On peut calculer approximativement le sinus d'un nombre x en effectuant la sommation des premiers termes de la série (une série est une somme infinie) :

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ...$$

où x est exprimé en radians et 3 ! désigne la factorielle de 3.

Écrire un programme qui lit une valeur flottante x en entrée et imprime une approximation de $sin(x)$.

Cette approximation sera obtenue en additionnant successivement les différents termes de la série jusqu'à ce que la valeur du terme devienne inférieure (en valeur absolue) à une constante ϵ que l'on fixera à 10^{-6}.

Exemple 1

Avec les données lues suivantes :

```
0.8
```

le résultat à imprimer vaudra :

```
0.7173557231746032
```

Remarque : Compte-tenu du manque de précision concernant les calculs sur les float, vous pourrez obtenir un résultat sensiblement différent. Ce n'est pas un problème, car UpyLaB acceptera toute réponse suffisamment proche de celle attendue, avec une tolérance de l'ordre de 1.0e-5.

Exemple 2

Avec les données lues suivantes :

```
-0.5
```

le résultat à imprimer vaudra :

```
-0.479425533234127
```

Consignes

— Nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
Vous avez besoin de calculer la factorielle d'un nombre ?

Comme nous proscrivons l'utilisation du module `math`, si vous avez besoin de calculer la factorielle d'un entier `n` (ce qui n'est pas certain), le code suivant calculer dans la variable fact, la valeur `n!` (factorielle de `n`) :

```
fact = 1
for i in range(2, n + 1):
   fact = fact * i
```

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Dans cet exercice, notre programme doit calculer une approximation du sinus d'un nombre mais il ne faut pas utiliser la fonction `sin` du module `math`.
— Pour éviter de calculer explicitement la valeur des factorielles, on pourra chercher à exprimer chacun des termes en fonction du précédent.
— Notez bien que les exposants de x ne sont que les nombres impairs, et que le signe alterne entre + et -.

3.5.14 Exercice UpyLaB 3.18 (Noir)

Énoncé

Écrire un code qui lit un nombre entier strictement positif `n` et imprime une pyramide de chiffres de hauteur `n` (sur `n` lignes complètes, c'est-à-dire toutes terminées par une fin de ligne).

— La première ligne imprime un "1" (au milieu de la pyramide).
— La ligne *i* commence par le chiffre *i % 10* et tant que l'on n'est pas au milieu, le chiffre suivant a la valeur suivante *((i+1) % 10)*.
— Après le milieu de la ligne, les chiffres vont en décroissant modulo 10 (symétriquement au début de la ligne).

Notons qu'à la dernière ligne, aucune espace n'est imprimée avant d'écrire les chiffres `0123...`.

Exemple 1

Avec la donnée lue suivante :

```
1
```

le résultat à imprimer vaudra :

```
1
```

Exemple 2

Avec la donnée lue suivante :

```
2
```

le résultat à imprimer vaudra :

```
 1
232
```

Exemple 3

Avec la donnée lue suivante :

```
10
```

le résultat à imprimer vaudra :

```
         1
        232
       34543
      4567654
     567898765
    67890109876
   7890123210987
  890123454321098
 90123456765432109
0123456789876543210
```

Consignes

— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu.
— En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Pour tester votre code, UpyLaB va l'exécuter plusieurs fois en lui fournissant à chaque test des nombres différents en entrée. Il vérifiera alors que le résultat affiché par votre code correspond à ce qui est attendu. N'hésitez donc pas à tester votre code en l'exécutant plusieurs fois dans votre IDE (Thonny ou PyCharm par exemple) avec des valeurs différentes en entrée y compris supérieure à 10.

3.6 L'instruction pass et quiz de fin de module

3.6.1 L'instruction pass

L'INSTRUCTION PASS QUI NE FAIT RIEN

Avant de terminer ce module, nous voudrions vous présenter l'instruction `pass`. L'instruction `pass` est une instruction simple, même très simple.

Sa syntaxe unique est

```
pass
```

et elle a comme effet, quand l'interpréteur l'exécute, de ne rien faire.

Dans ce cas, à quoi bon introduire une telle instruction ?

En fait, le programmeur utilise généralement l'instruction `pass` quand il est en phase de développement de son programme, et qu'il n'a pas encore écrit certaines parties de son code.

Par exemple ayant une certaine valeur pour `x`

```
if x < 0 :
   pass  # TODO compléter le code (cas où x < 0)
else :
   print('traitement du cas où x est positif')
```

l'instruction `pass`, éventuellement avec un commentaire du type `A_FAIRE` (ou `TO DO` en anglais), permet d'avoir un code syntaxiquement correct tout en soulignant qu'une partie du code reste à écrire.

3.6.2 Quiz de fin de module

QUIZ POUR RÉCAPITULER CE QUE L'ON A VU DANS CE MODULE

Dans ce module nous avons appris comment fonctionne la plupart des instructions de contrôle de flux Python.

Pour bien assimiler la matière, analysons quelques bouts de code pour déterminer ce qu'ils font.

QUIZ

Note : Voir le quiz de la section 3.6.2

3.7 Bilan du module

3.7.1 Qu'avons-nous vu dans ce module ?

BILAN EN BREF

Note : Voir la vidéo de la section 3.7.1 : Bilan du module

BILAN DU MODULE

Nous voici à la fin de ce module.

Nous y avons vu principalement comment contrôler la séquence des instructions en fonction de conditions. De façon plus précise, nous avons vu :

- que les conditions sont exprimées sous forme de valeurs et d'expressions booléennes qui après évaluation peuvent avoir une valeur vraie (`True` en anglais) ou fausse (`False` en anglais)
- que les opérateurs relationnels comparent les valeurs : <, <=, >, >=, == et `!=`
- que l'on peut connecter des expressions booléennes avec les opérateurs booléens `and`, `or` et `not` ; nous avons vu leur syntaxe et leur sémantique ainsi que les lois de De Morgan permettant de simplifier certaines expressions booléennes
- l'instruction conditionnelle `if` avec possibilité de parties `elif` et d'une partie `else`
- l'instruction `pass`
- l'instruction répétitive `while` appelée également boucle `while`
- la fonction prédéfinie `range` qui permet de générer une séquence de nombres entiers
- l'instruction répétitive `for` qui traite chaque élément d'une séquence

Nous avons également illustré tous ces concepts avec de nombreux exemples et la réalisation d'exercices supervisés ou réalisés de façon autonome avec UpyLaB.

En route pour le module suivant qui va vous montrer comment définir vos propres fonctions Python.

CHAPITRE 4

Les fonctions : créons les outils que nous voulons

4.1 Au menu : comment fonctionne une fonction ?

4.1.1 Présentation du menu de ce module

MENU DE CE MODULE : EN BREF

Python permet de découper les programmes grâce à des fonctions. Dans ce module nous allons vous expliquer comment définir et utiliser une fonction dans un code Python et comment écrire des programmes modulaires, ce qui signifie ici des programmes dont le code est découpé en parties avec des fonctions. Comme dans le module précédent, nous passerons de sections théoriques d'introduction à des sections de mise en pratique des concepts, d'abord supervisées et ensuite que vous réaliserez de façon autonome.

MENU DE CE MODULE

Note : Voir la vidéo de la section 4.1.1 : Les fonctions

4.2 Les fonctions prédéfinies et définies

4.2.1 Utilisation et définition de fonctions

FONCTIONS PRÉDÉFINIES

Note : Voir la vidéo de la section 4.2.1 : Les fonctions prédéfinies et leur utilisation

CONTENU DE LA VIDÉO

La vidéo précédente présente plusieurs fonctions Python prédéfinies, utilisables directement par l'interpréteur ou après leur importation par exemple des modules `math`, `random` ou `time`.

Code sur la console réalisé dans la vidéo

```
>>> x = 666
>>> y = 7
>>> z = 3.14159
>>> abs(x)
666
>>> abs(-10)
10
>>> dir(x)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__',
 ↪'__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__
 ↪ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__
 ↪init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul_
 ↪_', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__
 ↪rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod_
 ↪_', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '_
 ↪_rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__
 ↪', '__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes
 ↪', 'imag', 'numerator', 'real', 'to_bytes']
>>> divmod(x, y)
(95, 1)
>>> float(x)
666.0
>>> input()
>? 1111
'1111'
>>> int(z)
3
>>> max(x, y, 1515)
1515
>>> min(x, 1551, y+3, 27)
10
>>> round(z, 3)
3.142
>>> type(z)
<class 'float'>
>>> help(divmod)
Help on built-in function divmod in module builtins:

divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.

>>> divmod(x, y)
(95, 1)
>>> 666 // 7
95
>>> 666 % 7
1
>>> print("x vaut:", x)
x vaut: 666
>>> print()

>>> from math import pi, cos, log
>>> cos(pi/4)
```

(suite sur la page suivante)

(suite de la page précédente)

```
0.7071067811865476
>>> log(1024,2)
10.0
>>> help(log)
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.

>>> log(3)
1.0986122886681098
>>> log(100,10)
2.0
>>> import random
>>> random.randint(1, 100)
77
>>> import time
>>> time.sleep(5)

>>>
```

La vidéo en bref

Nous illustrons l'utilisation des fonctions prédéfinies `abs`, `dir`, `divmod`, `float`, `input`, `int`, `max`, `min`, `round`, `type`, `help` et `print`.

Nous montrons ce que réalisent certaines fonctions ou valent certaines constantes des modules `math` (`log`, `cos`, `pi`), `random` (`randint`) et `time` (`sleep`).

4.2.2 Définition de nouvelles fonctions

DÉFINITION DE NOUVELLES FONCTIONS

Note : Voir la vidéo de la section 4.2.2 : Définition d'une nouvelle fonction

CONTENU DE LA VIDÉO

La vidéo précédente illustre, à partir de l'exemple du pgcd, comment on peut créer ses propres fonctions en Python.

Script réalisé dans la vidéo

```
def pgcd(x, y):
    """Calcule le pgcd de 2 entiers x et y positifs"""
    while (y > 0):
        x, y  =  y, x % y
    return x

print('le pgcd de 112 et de 30 vaut : ', pgcd(112, 30))
a = int(input(" a = "))
```

(suite sur la page suivante)

(suite de la page précédente)

```
if pgcd(a, 6) == 2:
   print("le pgcd est égal à 2")
else:
   print("le pgcd est différent de 2")
```

La vidéo en bref

Nous avons présenté la syntaxe d'une définition de fonction sur l'exemple de la fonction `pgcd` : avec la ligne d'entête suivie d'un docstring expliquant ce que reçoit la fonction comme paramètres et ce qu'elle fait et du corps de la fonction terminée par l'instruction `return`.

Après avoir défini la fonction `pgcd`, la suite du code peut en faire usage.

La séquence d'exécution lors de la définition et de l'appel à une fonction est détaillée dans l'unité suivante.

4.2.3 Exécution de fonctions

SÉQUENCE D'EXÉCUTION D'UNE FONCTION

Note : Voir la vidéo de la section 4.2.3 : Exécution d'une fonction

CONTENU DE LA VIDÉO

La vidéo précédente présente la séquence d'exécution lors de l'appel à une telle fonction Python.

Script réalisé dans la vidéo

```
def pgcd(x, y):
    """Calcule le pgcd de 2 entiers x et y positifs"""
    while (y > 0):
       x, y  =  y, x % y
    return x

print('le pgcd de 112 et de 30 vaut : ', pgcd(112, 30))
a = int(input(" a = "))
if pgcd(a, 6) == 2:
   print("le pgcd est égal à 2")
else:
   print("le pgcd est différent de 2")
```

La vidéo en bref

Lorsqu'une fonction est appelée par une autre fonction ou dans le code « principal » (que nous appelons le code « appelant »), la séquence du traitement est la suivante :

1) Les variables locales à la fonction appelée, correspondantes aux paramètres formels, sont créées (et mises dans le nouvel « espace de nom » associé à cette instance de la fonction).
2) Les paramètres sont transmis (passés) entre le code appelant et la fonction appelée. Ceci est décrit plus en détails à la sous-section suivante.
3) L'exécution du code appelant est suspendue pour laisser la place à l'exécution de la fonction appelée.

4) La fonction appelée s'exécute éventuellement en créant de nouvelles variables et des objets.
5) Lorsque la fonction appelée termine son exécution soit quand une instruction `return` est exécutée (n'importe où dans la fonction), soit après l'exécution de la dernière instruction de la fonction, les variables locales à la fonction sont "détruites" : plus précisément l'espace de nom associé à l'instance de la fonction est détruit.
6) Le code appelant reprend son exécution. Si la fonction précédemment appelée était une fonction retournant une valeur avec l'instruction `return`, la valeur de retour a été conservée pour permettre l'évaluation de l'expression utilisant le résultat de cette fonction.

4.2.4 Précisions sur les fonctions

PRÉCISIONS SUR LES FONCTIONS ET LE MÉCANISME DE PASSAGE DE PARAMÈTRES

Notons qu'un *espace de nom* (Namespace) est l'endroit où des variables locales sont stockées, sachant qu'en Python une variable est un nom pour un objet.

Il doit y avoir une correspondance entre les paramètres formels et les paramètres effectifs selon la position. Cela signifie que le i-ème paramètre effectif en ordre d'apparition dans la liste doit correspondre au i-ème paramètre formel.

Le passage de paramètres suit le même fonctionnement qu'une assignation : un paramètre effectif soit désigne un objet existant soit est une expression dont la valeur donne un objet ; le paramètre formel correspondant est une variable locale à la fonction qui lors de son exécution, sera un nom local de cet objet.

Donc avec `def pgcd(x, y)` : l'appel `pgcd(26, a)` aura comme effet que dans l'instance de la fonction `pgcd`, `x` désigne l'objet entier valant `26` et `y`, la valeur de la variable nommée par ailleurs `a`.

Le mot instance de fonction signifie exemplaire ; en effet une fonction peut être appelée plusieurs fois, et peut même s'appeler elle même « récursivement » ce qui entraîne plusieurs instances simultanées lors de l'exécution. La récursivité ne sera pas abordée dans ce cours.

COMMENT ET POURQUOI DÉFINIR ET UTILISER DES FONCTIONS ?

Pour définir une nouvelle fonction, il faut écrire son **entête** qui commence par le mot-clé « def » suivi du nom de la fonction et des paramètres appelés dans le jargon standard **paramètres formels** entourés de parenthèses, le tout terminé par le caractère deux-points (« : »).

Le **corps** de la fonction est identifié par l'ensemble des lignes de code qui suivent indentées par rapport à l'entête.

```
def nom (paramètres formels):
    corps de la fonction
```

L'utilisation de la fonction se fait quand une instruction donne le nom de cette fonction avec les **paramètres effectifs**, également appelés **paramètres réels** ou **arguments**, correspondants.

```
nom(arguments)
```

Chaque fonction peut être vue comme un outil résolvant un certain problème. Les autres fonctions peuvent alors utiliser cet outil sans se soucier de sa structure, juste pour le résultat qu'il donne. Pour pouvoir l'utiliser, il leur suffit de savoir comment l'employer, c'est-à-dire de connaître le nom de la fonction, les valeurs qu'elle nécessite et la façon dont elle renvoie un ou plusieurs résultats.

Cette technique a, en particulier, deux avantages :
- chaque fonction peut (généralement) être testée indépendamment du reste du programme ;
- si une partie contient une séquence d'instructions qui doit être réalisée à différents endroits du programme, il est possible de n'écrire cette séquence qu'une seule fois.

En gros, une fonction peut être vue comme une opération dont la valeur est définie par le programmeur en fonction des paramètres reçus.

MAIS AU FAIT, COMMENT LA FONCTION RENVOIE-T-ELLE SON RÉSULTAT ?

Nous avons dit que la fonction renvoie le résultat de son travail. Mais comment ? Par une petite instruction bien particulière nommée return. Souvenez-vous de la définition de notre fonction pgcd :

```
def pgcd(x, y):
    """Calcule le pgcd de 2 entiers x et y positifs"""
    while (y > 0):
        x, y  =  y, x % y
    return x
```

C'est parce qu'il y a cette instruction `return x` que l'on peut effectuer l'affectation suivante :

```
>>> ma_valeur = pgcd(142, 36)
```

Attention, dès que l'exécution de votre code rencontre un `return`, la fonction en question se termine. Ainsi, toute instruction se trouvant après le return sera tout simplement ignorée. Considérons cette fonction :

```
def foo(x):
    if x > 0:
        return math.sqrt(x)
    print('Fini')
```

Alors l'appel suivant n'imprime pas le texte `Fini` à l'écran :

```
>>> foo(10)
3.1622776601683795
```

Explications : lors de l'appel, la variable locale `x` vaut `10`, le test est donc vrai et le code exécute l'instruction `return` qui quitte la fonction. Le `print` n'est donc pas atteint. Par contre, ce deuxième appel affiche bien `Fini` à l'écran :

```
>>> foo(-1)
Fini
```

FONCTION : PRINT ET INSTRUCTION RETURN : ATTENTION À LA CONFUSION

Revenons sur notre fonction `pgcd` et donnons en une version modifiée :

```
def pgcd2(x, y):
    """Calcule le pgcd de 2 entiers x et y positifs"""
    while (y > 0):
        x, y  =  y, x % y
    print(x)
```

Dans un interprète interactif, faisons quelques tests :

```
>>> pgcd(112, 30)
2
>>> pgcd2(112, 30)
2
```

Il semble ne pas y avoir de différence... et pourtant il y en a une et de taille. Informellement, la version originale, avec l'instruction `return`, nous renvoie la valeur calculée du pgcd. Dans la seconde version, elle ne fait que nous la montrer, sans que nous puissions en disposer. Lors des appels dans l'interprète interactif, le premier appel renvoie à l'interprète la valeur calculée 2 et c'est l'interprète qui nous affiche cette valeur. Dans le second appel c'est la fonction qui réalise l'affichage par l'appel à la fonction `print`. Ainsi :

```
>>> x = pgcd(112, 30)
>>> x
2
```

Mais,

```
>>> x = pgcd2(112, 30)
2
>>> x
None
```

4.2.5 Mini quiz

MINI QUIZ SUR LES FONCTIONS

Avant d'examiner de plus près les possibilités et le fonctionnement des fonctions Python, ce quiz va vous aider à vérifier que vous avez bien assimilé ce que nous avons vu jusqu'ici.

Note : Voir le quiz de la section 4.2.5 du cours en ligne

4.3 Faisons fonctionner les fonctions

4.3.1 Autres exemples de fonctions

AUTRES EXEMPLES DE FONCTIONS AVEC PARAMÈTRES

Donnons d'autres exemples de fonctions, qui varient en particulier selon les paramètres reçus et le ou les résultat(s) renvoyé(s) :

Fonction booléenne : La fonction `est_pair` est une fonction booléenne, c'est-à-dire dont le résultat est une valeur booléenne, qui renvoie *vrai* (`True`) si la valeur reçue en paramètre est paire.

```
def est_pair(x):
    """Renvoie vrai si et seulement si x est pair."""
    return x % 2 == 0

a = int(input("Entrez un nombre entier impair : "))
if est_pair(a):
    print("Il n'est pas impair !")
```

Fonction qui renvoie None : La fonction `print_line` affiche un résultat sur l'écran (l'output) mais ne renvoie pas de valeur.

```
def print_line(x, n):
    """Imprime x*n."""
    print(x*n)
```

Par défaut, la fonction renvoie la valeur `None`. Il s'agit d'une valeur d'un type à part (`NoneType`) qui symbolise l'absence de valeur. Cela signifie que l'instruction `return` sans rien derrière équivaut à `return None`. Par ailleurs, un `return` implicite existe à la fin de toute fonction pour revenir au code appelant (avec la valeur de retour `None`).

Fonction qui renvoie un tuple : La fonction `min_et_max` renvoie deux valeurs :

```
def min_et_max(x, y):
    """Renvoie min(x, y) suivi de max(x, y)."""
    if x > y:
        x, y = y, x
    return x, y
```

Plus précisément, toute fonction Python renvoie une seule valeur à la fin de son exécution. Ici `min_et_max` renvoie en réalité un objet de type tuple qui contient deux valeurs. Nous verrons dans le prochain module, qui traite des séquences, cette notion de tuple plus en détails.

UN EXEMPLE DE FONCTION SANS PARAMÈTRE

Fonction sans paramètre : Notons que nous pouvons définir une fonction sans paramètre. Dans ce cas, dans l'entête de cette dernière, il faut quand même mettre des parenthèses sans rien dedans. Par exemple, le code suivant utilise `turtle`, et en particulier la fonction `circle`, pour dessiner un symbole Yin et yang. Pour ce faire, nous utilisons la fonction prédéfinie `turtle.circle` avec un ou deux paramètres :

turtle.circle(rayon [, angle]) :

- trace un cercle ou un arc de cercle de |*rayon*| (valeur absolue de rayon) ;
- le cercle est complet si le paramètre angle n'existe pas ;
- le cercle ne fait qu'un arc de `angle` degrés si `angle` est donné.

Par exemple `turtle.circle(100,180)` trace un demi cercle (soit 180 degrés) de rayon de 100 points :

- si `rayon` est positif, le cercle est tracé dans le sens positif, c'est-à-dire anti-horlogique, sinon (`rayon` est négatif), il est tracé dans le sens horlogique ;
- si `angle` est positif, la tortue avance, sinon elle recule.

Le code ci-dessous donne une solution à la fonction `yin_yang()` :

```
""" Exemple de code sans paramètre :
yin et yang dessiné avec turtle
"""
import turtle

def yin_yang():
    """ dessine un logo yin-yang de rayon 200 """
    turtle.down()  # met la plume en mode tracé (si ce n'était déjà le cas)
    turtle.width(2)  # grosseur du tracé de 2 points
    # dessine le yin externe
    turtle.color("black", "black")  # le tracé et le remplissage seront en noir
    turtle.begin_fill()  # la ou les formes suivantes seront remplies
    turtle.circle(-100, 180)  # demi cercle intérieur tournant vers la droite
    turtle.circle(-200, -180)  # demi cercle extérieur, en marche arrière
    turtle.circle(-100, -180)  # demi cercle intérieur qui complète le yin
    turtle.end_fill() # remplissage
    # dessine le yang interne
    turtle.color("white") # couleur blanche
    turtle.up() # on ne trace pas ce qui suit
    # déplace la tortue au bon endroit
    turtle.right(90)
    turtle.forward(80)
    turtle.left(90)
    # tracé du disque yang (blanc) interne au yin
    turtle.down()
    turtle.begin_fill()
    turtle.circle(-20)
    turtle.end_fill()
    # se replace au centre
    turtle.up()
    turtle.left(90)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    turtle.forward(80)
    turtle.right(90)
    # dessine le yang externe
    turtle.down()
    turtle.color("black", "white") # contour noir, remplissage blanc
    turtle.begin_fill()
    turtle.circle(-100, 180)
    turtle.circle(-200, -180)
    turtle.circle(-100, -180)
    turtle.end_fill()
    # tracé du disque yin (noir) interne au yang

    turtle.color("black")
    # déplace la tortue au bon endroit
    turtle.up()
    turtle.right(90)
    turtle.forward(80)
    turtle.left(90)
    turtle.down()
    # trace le disque
    turtle.begin_fill()
    turtle.circle(-20)
    turtle.end_fill()
    # se replace au centre
    turtle.up()
    turtle.left(90)
    turtle.forward(80)
    turtle.right(90)
    turtle.down()
    turtle.hideturtle()
    return

#code principal
yin_yang() #réalise le logo
```

L'exécution du code ci-dessus donne le résultat montré par la petite vidéo donné en section 4.3.1 du cours en ligne.

DESSIN D'UN YIN-YANG AVEC TURTLE

Note : Voir la vidéo de la section 4.3.1 du cours en ligne : Dessin d'un yin-yang avec turtle

COMPLÉMENT D'INFORMATIONS

Notons dans le code précédent l'utilisation de `turtle.color(couleur1, couleur2)` qui demande que les tracés aient les contours en couleur1 (`"black"` par exemple) et les surfaces remplies (grâce à `turtle.begin_fill()` et `turtle.end_fill()`) en couleur2 (`"white"` par exemple pour faire le yang).

Nous discuterons de la qualité de ce code dans la section suivante qui parle de règles de bonnes pratiques pour coder ; nous y verrons pourquoi un programmeur expérimenté n'aurait pas écrit la fonction `yin_yang` telle quelle.

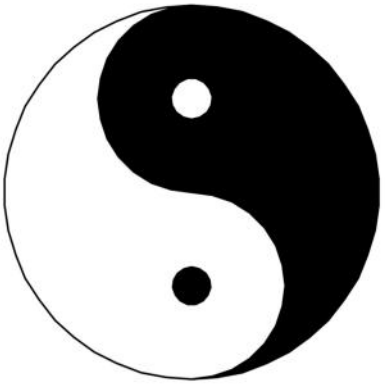


Fig. 4.1 – Dessin de yin-yang est réalisé avec turtle

4.3.2 Quelques aspects plus techniques

Avant de mettre en pratique les fonctions, les points suivants, plus techniques, doivent être mis en évidence car ils sont importants pour ne pas écrire de codes erronés.

POLYMORPHISME ET CONTRÔLE DE TYPE

Même si en pratique ce n'est généralement pas le cas, en théorie une fonction Python peut recevoir différents types de paramètres et renvoyer des résultats de types différents : c'est ce que l'on appelle la surcharge des paramètres.

Par exemple avec la définition :

```
def sum(a,b):
    """renvoie la somme ou concaténation des deux paramètres"""
    return a+b
```

les 3 appels successifs :

```
print(sum(3,5))
print(sum('cha', 'peau'))
print(sum(3, 'peau'))
```

renvoient respectivement :

```
8
chapeau
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 2, in sum
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

La fonction `sum` peut donc faire des sommes de valeurs, des concaténations de texte (qui consistent à coller deux textes ensemble) ou même produire des erreurs, suivant le type des arguments utilisés. Dans le jargon informatique, on parle de polymorphisme, ce qui signifie que quelque chose, ici une fonction, de même forme, peut en fait représenter plusieurs choses ou comme ici, avoir des effets différents.

En Python, il est donc généralement fort conseillé, si l'on n'est pas sûr du code appelant, que la fonction vérifie elle-même que les arguments donnés aient un type adéquat pour son exécution.

La section suivante explique une bonne façon de faire cela.

ASSERT

L'instruction `assert` vérifie si une certaine condition est rencontrée. Dans l'exemple précédent, si les seuls types permis pour les paramètres de notre fonction `sum` sont des `int` ou des `float`, une instruction à ajouter en début du code de la fonction juste après l'entête et le docstring de la fonction est :

```
assert (type(a) is int or type(a) is float) and (type(b) is int or type(b) is float)
```

qui, lors de son exécution, teste la condition ; si elle est vraie, l'interpréteur passe à l'instruction suivante, sinon il provoque une erreur qui indique que l'assertion est fausse.

VARIABLES GLOBALES ET LOCALES

Dans la section précédente, nous avons vu que lors de son exécution, la fonction appelée utilise des variables qui lui sont locales. Nous avons vu comment s'effectue le passage de paramètres entre les arguments effectifs et les paramètres formels. Par ailleurs, le code principal utilise des variables globales que l'interpréteur continue à manipuler après l'exécution de la fonction. La présente unité revient en détails sur les mécanismes qui gèrent les variables : en effet, nous devons bien les comprendre, sous peine d'écrire des codes qui ne font pas du tout ce que nous désirons. Allons-y !

Assignation : Nous avons vu en section 2.7 qu'après les instructions :

```
a = b = 3
b = 2 * b
```

la variable `b` vaut `6` tandis que la variable `a` vaut toujours `3`.

Plus précisément, après la première instruction, `a` et `b` sont deux noms pour l'objet entier valant `3`, comme le montre le premier diagramme d'état ci-dessous. Ensuite, l'évaluation de `2 * b` crée un nouvel objet entier valant `6` et `b` référence `6` comme le montre le second diagramme d'état après la seconde instruction.

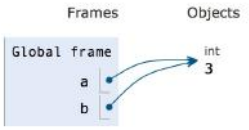


Fig. 4.2 – Diagramme d'état après `a = b = 3`

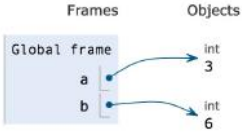


Fig. 4.3 – Diagramme d'état après `b = 2 * b`

Pour vous en convaincre complètement, vous pouvez exécuter étape par étape l'exemple avec l'animation Python Tutor suivante :

Note : Voir la première animation Python Tutor en section 4.3.2 du cours en ligne

Exécution étape par étape du code avec Python Tutor

Le passage de paramètres utilise le même mécanisme comme le montre l'exemple suivant :

```
def fun(x):
    x = x**2
    return x
```

(suite sur la page suivante)

(suite de la page précédente)

```
a = 3
fun(a)
print(a)
```

On peut voir que la variable `a` n'a pas été modifiée par la fonction comme le montre le diagramme d'état au moment du `return`.

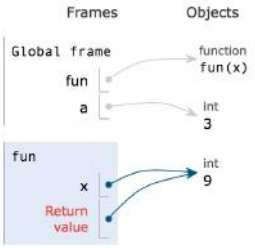


Diagramme d'état au moment du `return x`

À nouveau pour vous en convaincre complètement, vous pouvez exécuter étape par étape l'exemple avec l'animation Python Tutor suivante :

Note : Exécution étape par étape du code avec Python Tutor : voir cours en ligne section 4.3.2

Nom des variables globales et locales

Même si il est mieux d'éviter cela pour des raisons de clarté du code global, nous pouvons vérifier que le fait qu'une variable locale porte le même nom qu'une variable globale ne change rien. Le code qui suit est une modification de l'exemple précédent mais manipule une variable globale `x` ainsi qu'une autre variable locale à la fonction `fun`, qui porte le même nom `x`.

```
def fun(x):
    x = x**2
    return x

x = 3
fun(x)
print(x)
```

Nous pouvons exécuter étape par étape l'exemple avec l'animation Python Tutor suivante :

Note : Voir la troisième animation Python Tutor en section 4.3.2 du cours en ligne

Les animations Python Tutor présentées précédemment montrent clairement qu'une variable locale, comme le paramètre formel `x` de la fonction `fun`, n'est pas connue en dehors de l'exécution de l'instance de fonction où elle est créée. De plus à la fin de l'exécution de cette instance de fonction, les variables locales à cette instance de fonction sont supprimées. Ce qui nous amène à la question suivante :

Pourquoi les fonctions Python utilisent-elles des variables locales ?

Quand le programmeur écrit une fonction `fun`, l'échange de valeurs et de résultats entre la fonction `fun` et la fonction (ou le programme global) est réalisé via les mécanismes du passage des paramètres et par l'instruction `return`. Pour que les codes de la fonction et du programme soient « propres », il faut éviter toute autre inférence : c'est ce qui est réalisé par le mécanisme des variables locales qui permet d'isoler proprement le code de la fonction `fun` (utilisant uniquement des variables locales qui lui sont propres) du code qui utilise cette fonction `fun` et d'autres variables.

Attention, jusqu'à présent, aucun type d'objets présenté dans ce cours n'est modifiable. En effet, par exemple

```
x = 1
x = 2
```

crée un premier objet de type int valant 1 nommé `x` via l'affectation et ensuite un second objet de même type valant 2 et le nom `x` est donné à ce second objet via la seconde affectation ; ce qui rend le premier objet inaccessible puisqu'il n'a plus de nom. Un nom de variable peut donc désigner des objets différents tout au long de l'exécution, mais dans cet exemple les objets eux-mêmes de type `int` sont non modifiables.

Nous verrons dans le prochain module que certains objets (par exemple les listes) sont modifiables. Dans ce cas, si c'est requis pour la fonction `fun`, le mécanisme de passage de paramètres permettra à la fonction de modifier un paramètre, c'est-à-dire un objet reçu du code appelant la fonction `fun`. Notez cependant que si aucune modification de paramètre n'est demandée, une règle de bonne pratique est de veiller à ne pas modifier ceux-ci dans le code de la fonction. Les exemples plus haut de fonction `fun` qui modifient le paramètre formel `x` sont donc à éviter.

Certains mécanismes permettent de déroger au mécanisme de variables locales et globales en permettant, par exemple, à une fonction de manipuler des variables globales. Comme ce n'est généralement pas utile ni une bonne pratique pour obtenir un code « propre », nous vous recommandons de ne pas utiliser ces mécanismes et de n'utiliser que des variables locales. Dans ce cours, seules des constantes globales seront utilisées, comme nous le verrons à la section suivante.

UN PEU DE VOCABULAIRE : PORTÉE ET VISIBILITÉ

Les variables qui sont créées et ensuite supprimées font appel à deux concepts : portée et index :*visibilité*. Expliquons ces termes utilisés fréquemment par les programmeurs.

Dans le jargon informatique, l'ensemble des endroits où dans un programme, une variable existe est appelé sa portée (scope en anglais). L'ensemble des endroits où une variable est visible est appelé sa visibilité.

Dans l'exemple précédent, une variable globale, comme `x`, existe depuis sa création jusqu'à la fin de l'exécution du programme, mais n'est pas visible dans la fonction `fun` : en l'occurrence la variable `x` locale cache la variable `x` globale.

TRACEBACK

Nous avons déjà rencontré le mot Traceback lorsque notre code s'arrêtait en produisant une erreur. Le Traceback est une information que l'interpréteur donne au programmeur pour qu'il trouve son erreur de code. Une traduction possible de ce verbe est retracer. C'est donc une trace de l'exécution au moment de l'erreur. Expliquons la notion de Traceback sur un exemple. L'exécution du code ci-dessous (où nous avons rajouté les numéros de ligne pour que ce soit plus clair), comme script PyCharm sauvé dans mon répertoire Desktop par exemple,

```
1  def ma_fun():
2      une_autre_fun()
3
4  def une_autre_fun():
5      encore_une_fun()
6
7  def encore_une_fun():
8      # une_erreur est utilisé ligne 9 sans avoir été définie !
9      une_erreur = une_erreur + 1
10
11 ma_fun()
```

génère le message d'erreur suivant quand l'interpréteur essaye d'exécuter l'instruction `une_erreur = une_erreur + 1` alors que la variable `une_erreur` n'est pas encore définie (aucune assignation n'a été effectuée avant).

```
Traceback (most recent call last):
    File "/Users/tmassart/Desktop/test_4_1_2018.py", line 11, in <module>
        ma_fun()
    File "/Users/tmassart/Desktop/test_4_1_2018.py", line 2, in ma_fun
```

(suite sur la page suivante)

(suite de la page précédente)

```
        une_autre_fun()
    File "/Users/tmassart/Desktop/test_4_1_2018.py", line 5, in une_autre_fun
        encore_une_fun()
    File "/Users/tmassart/Desktop/test_4_1_2018.py", line 9, in encore_une_fun
        une_erreur = une_erreur + 1
UnboundLocalError: local variable 'une_erreur' referenced before assignment
```

Le message d'erreur explique le type de l'erreur (en dernière ligne) tandis que les lignes précédentes donnent un résumé du « chemin » pris lors de l'exécution pour arriver à l'instruction qui a causé cette erreur. On peut ainsi retracer la séquence des appels en cours quand l'erreur s'est produite.

4.3.3 Première mise en pratique des fonctions

LE N-IÈME NOMBRE DE FIBONACCI

Après cette longue suite d'explications, il est grand temps de mettre en pratique les nouveaux concepts que nous avons vus dans ce module.

Reprenons le principe du calcul des nombres de Fibonacci présenté dans le module 3, section 3.5.1.

Rappelons que les premiers nombres de Fibonacci sont :

$$0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ 34\ldots$$

où après les deux premiers qui sont donnés, la valeur des nombres suivants est donnée en sommant les deux précédents.

Plus mathématiquement, nous pouvons dénoter les nombres de Fibonacci par :

$$F_0 = 0 \tag{4.1}$$
$$F_1 = 1 \tag{4.2}$$
$$F_{i+1} = F_{i-1} + F_i, \text{ pour } i > 0 \tag{4.3}$$

Supposons que notre programme utilise régulièrement des nombres de Fibonacci. Une solution simple est d'écrire une fonction qui calcule et renvoie F_n pour l'indice n désiré.

Commençons par choisir un nom à notre fonction et déterminer le ou les paramètre(s) formel(s).

Les règles de bonnes pratiques pour encoder des programmes Python, comme discutées dans la section suivante, demandent que les noms des fonctions, comme celui des variables, débutent par une lettre alphabétique en minuscule et soient formés de lettres alphabétiques minuscules, de chiffres ou du caractère souligné `'_'`. Dans le jargon, cette convention s'appelle « snake case ».

Appelons notre fonction `fibo` et le paramètre formel `n` qui correspond à l'indice du nombre de Fibonacci à calculer.

```
def fibo(n):
```

Ensuite, écrivons le docstring en indiquant ce que fait `fibo` et le contenu initial de `n` au moment de chaque appel :

```
"""calcule le n-ième nombre de Fibonacci, avec : n de type int et
fibo(0) valant 0
fibo(1) valant 1 et
fibo(n+1) valant fibo(n-1) + fibo(n)
si n < 0 : fibo(n) retourne None"""
```

Après, le corps de la fonction sera écrit et terminé par un `return` du résultat.

Il nous reste à :

- déterminer les hypothèses. Par exemple supposons, sans devoir le tester, que n est de type int ;
- identifier les différents cas (`n < 0`, `n == 0`, `n == 1`, `n > 1`) ;
- déterminer ce que la fonction renvoie dans chacun des cas.

Enfin, **après** cette définition de la fonction `fibo(n)`, nous pouvons écrire le code qui utilise cette fonction.

À VOUS DE JOUER !

Ouvrez un nouveau script dans PyCharm et complétez le code pour avoir une définition complète de la fonction `fibo(n)`.

Un conseil : pour le cas général, une instruction répétitive pourrait être utile.

Ensuite, en dessous de la définition, rajoutez un code qui teste la fonction, par exemple en imprimant le résultat pour les valeurs entre 0 et 100 non compris.

Votre code aura donc la structure suivante :

```
def fibo(n):
    """calcule le n-ième nombre de Fibonacci, avec : n de type int et
    fibo(0) valant 0
    fibo(1) valant 1 et
    fibo(n+1) valant fibo(n-1) + fibo(n)
    si n < 0 : fibo(n) retourne None"""
    ...
    code correspondant au corps de la fonction fibo
    ...
    return res

for i in range(100):
    print(fibo(i))
```

Si vous ne trouvez pas

Vous n'arrivez pas à réaliser l'exercice. Voici une solution pour le résoudre :

```
def fibo(n):
    """calcule le n-ième nombre de Fibonacci, avec : n de type int et
    fibo(0) valant 0
    fibo(1) valant 1 et
    fibo(n+1) valant fibo(n-1) + fibo(n)"""
    if n < 0 : # fibo(n) retourne None
        res = None
    else:
        res = 0
        suivant = 1
        for _ in range(n): # calcule n fois le nombre de Fibonacci suivant
            res, suivant = suivant, res + suivant
    return res

for i in range(100):
    print(fibo(i))
```

POUR ALLER PLUS LOIN : LA RÉCURSIVITÉ

Version de fibo récursive

La programmation de la fonction `fibo` peut se faire en utilisant une technique algorithmique plus évoluée : la récursivité. Même si la récursivité ne fait pas partie de la matière abordée dans ce cours, pourquoi ne pas être curieux et chercher avec votre navigateur `récursivité` ?

En pratique une fonction est récursive si (directement ou indirectement) elle s'appelle elle-même !

Ainsi une version récursive simple (ici sans docstring et qui, pour simplifier, suppose que n vaut au moins 0) de la fonction fibo pourra s'écrire :

```
def fibo(n):
    if n < 2:
        res = n                         # fibo(0) vaut 0 et fibo(1) vaut 1
    else:                               # n >= 2
        res = fibo(n-1) + fibo(n-2)
    return res
```

Notez que si une solution récursive simple est très concise, dans ce cas-ci, sans technique plus élaborée comme la programmation dynamique (non vue dans ce cours), elle n'est pas une bonne idée car pas efficace en temps d'exécution.

4.3.4 Mise en pratique des fonctions : exercices UpyLaB 4.1 et suivants

MISE EN PRATIQUE AVEC UPYLAB

Faites les exercices UpyLaB 4.1 à 4.3 qui suivent.

EXERCICE UPYLAB 4.1 (Vert)

Énoncé

Écrire une fonction `deux_egaux(a, b, c)` qui reçoit trois nombres en paramètre et qui renvoie **la valeur booléenne** `True` si au moins deux de ces nombres ont la même valeur, et **la valeur booléenne** `False` sinon.

Ensuite, **écrire un programme** qui lit trois données de type int, `x`, `y` et `z`, et affiche le résultat de l'exécution de `deux_egaux(x, y, z)`.

Exemple 1

Avec les données lues suivantes :

```
1
2
3
```

le résultat à imprimer vaudra donc

```
False
```

Exemple 2

Avec les données lues suivantes :

```
42
6
42
```

le résultat à imprimer vaudra donc

```
True
```

Consignes

- Dans cet exercice, il vous est demandé d'écrire une fonction, puis un programme appelant cette fonction sur des valeurs lues en entrée. Notez qu'UpyLaB testera ces deux points, en exécutant le programme entier mais aussi en appelant directement la fonction avec les arguments de son choix.
- Plus précisément UpyLaB testera d'abord l'existence d'une définition de la fonction `deux_egaux` avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre programme en ajoutant 2 appels à la fonction `deux_egaux` pour vérifier que celle-ci n'effectue aucun `print`. C'est en effet à votre programme d'effectuer le `print` demandé. Ensuite différents tests de votre programme complet et de la fonction seront effectués par UpyLaB.
- Il n'est pas demandé que la fonction `deux_egaux` teste le type des paramètres reçus.
- Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).
- Par contre quand UpyLaB teste spécifiquement le code d'une fonction (ici la fonction `deux_egaux`), le type (et on verra plus tard aussi la structure) du résultat est égaement validé. Veillez donc bien à renvoyer un résultat de type requis.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Veillez à ce que la fonction retourne bien les valeurs booléennes `True` ou `False`, et non les chaînes de caractères `"True"` ou `"False"`.
- Pour retourner une valeur, une fonction doit utiliser l'instruction `return valeur` et non pas l'instruction `print`.
- La fonction doit retourner `True` si au moins deux nombres passés en paramètre sont égaux, que doit alors retourner l'appel `deux_egaux(2, 2, 2)` ?
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.1*.

4.3.5 Exercice UpyLaB 4.2 (Vert)

Énoncé

Attention : cet exercice est composé de l'exercice UpyLaB 4.2.a suivi en dessous de l'exercice UpyLaB 4.2.b.

Le Petit Prince vient de débarquer sur la planète U357, et il apprend qu'il peut y voir de belles aurores boréales !

La planète U357 a deux soleils : les étoiles E1515 et E666. C'est pour cela que les tempêtes magnétiques sont permanentes, ce qui est excellent pour avoir des aurores boréales.

Par contre, il y fait souvent jour sauf bien évidemment quand les deux soleils sont couchés en même temps.

Heureusement pour nous, une journée U357 s'écoule sur 24 heures comme sur notre Terre, et pour simplifier, nous ne prendrons pas en compte les minutes (on ne donne que les heures avec des valeurs entières entre 0 et 23).

Nous vous demandons d'aider le Petit Prince à déterminer les périodes de jour et de nuit.

UPYLAB 4.2A

Pour cela, vous allez dans un premier temps écrire une fonction `soleil_leve` qui, pour un soleil particulier, reçoit trois valeur

- l'heure de lever du soleil (entre 0 et 23)
- l'heure du coucher du soleil (entre 0 et 23)
- l'heure actuelle (entre 0 et 23)

et qui renvoie une valeur booléenne vraie si le soleil est levé sur la planète à l'heure donnée en troisième argument et fausse, s'il est couché.

On supposera que chacun des soleils ne se lève et ne se couche au plus qu'une seule fois par jour. Il est toutefois possible que le lever ait lieu après l'heure du coucher, ce qui signifie dans ce cas que le soleil est levé au début de la journée, puis qu'il se couche, puis qu'il se lève à nouveau plus tard dans la journée. Enfin, si l'heure du lever est la même que l'heure du coucher :

- soit toutes deux valent 12, cela signifie que le soleil ne se lève pas de la journée,
- soit toutes les deux valent 0, cela signifie que le soleil ne se couche pas de la journée.

Exemple 1

L'appel de la fonction suivant :

```
soleil_leve(6, 18, 10)
```

doit retourner

```
True
```

Exemple 2

L'appel de la fonction suivant :

```
soleil_leve(15, 8, 12)
```

doit retourner

```
False
```

Exemple 3

L'appel de la fonction suivant :

```
soleil_leve(12, 12, 10)
```

doit retourner

```
False
```

Exemple 4

L'appel de la fonction suivant :

```
soleil_leve(0, 0, 22)
```

doit retourner

```
True
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire la fonction `soleil_leve`. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Plus précisément UpyLaB testera d'abord l'existence d'une définition de fonction `soleil_leve` avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre fonction en exécutant un code qui réalise un appel à votre fonction et affiche le résultat, ceci pour vérifier que celle-ci n'effectue aucun print. Ensuite différents tests de votre fonction seront effectués par UpyLaB.
— Il n'est pas demandé que la fonction `soleil_leve` teste le type du paramètre reçu.
— Quand UpyLaB teste spécifiquement le code d'une fonction (ici la fonction `soleil_leve`), le type du résultat est également validé. Veillez donc bien à renvoyer un résultat de type requis. En particulier, les objets `True` et `"True"` ne sont pas du même type.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une solution à l'exercice : **`vidéo d'une solution de l'exercice UpyLaB 4.2a et 4.2b<https ://pod.univ-reunion.fr/video/2567-replay-du-live-du-25-mars-2020/>`_**

UPYLAB 4.2B

Il vous faut maintenant **écrire un programme** qui lit en entrée :

. l'heure de lever du soleil E1515 . l'heure du coucher du soleil E1515 . l'heure de lever du soleil E666 . l'heure du coucher du soleil E666

et qui utilise la fonction `soleil_leve` pour afficher ligne par ligne chacune des heures de la journée, depuis 0 jusqu'à 23, suivies d'une espace et d'une astérisque s'il fait nuit à cette heure.

Attention, il ne fera nuit que si E1515 et E666 sont tous deux couchés.

Exemple 1

Avec les données lues suivantes :

```
6
18
10
21
```

le résultat à imprimer vaudra donc

```
0 *
1 *
2 *
3 *
4 *
5 *
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21 *
22 *
23 *
```

Exemple 2

Avec les données lues suivantes :

```
15
8
6
17
```

le résultat à imprimer vaudra donc

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
```

(suite sur la page suivante)

(suite de la page précédente)

```
22
23
```

Consignes

— N'oubliez pas d'insérer votre fonction `soleil_leve`.
— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une solution à l'exercice : **`vidéo d'une solution de l'exercice UpyLaB 4.2a et 4.2b<https ://pod.univ-reunion.fr/video/2567-replay-du-live-du-25-mars-2020/>`_**

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Veillez à ce que la fonction retourne bien les valeurs booléennes `True` ou `False`, et non les chaînes de caractères `"True"` ou `"False"`.
— Pour retourner une valeur, une fonction doit utiliser l'instruction `return valeur` et non pas l'instruction `print`.
— La difficulté réside dans l'identification de tous les cas possibles. N'hésitez donc pas à tester votre fonction dans votre IDE ou dans PythonTutor avec plusieurs jeux de valeurs.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.2*.

4.3.6 Exercice UpyLaB 4.3 (Bleu)

CALCULS DE NOMBRES PREMIERS : FONCTION PREMIER(N)

L'exercice 4.3 d'UpyLaB vous propose une fonction assez classique à rédiger, qui teste si un nombre `n`, reçu en paramètre, est premier.

Par définition, un nombre entier naturel est premier s'il est divisible (entièrement) uniquement par deux nombres différents, par 1 et par lui-même. 1 n'est pas premier. Le plus petit nombre premier est 2. Tous les nombres premiers suivants sont impairs puisque tous les nombres pairs sont divisibles par 2.

Il faut rédiger une fonction booléenne (dont le résultat est `True` ou `False`), qui reçoit un entier positif et renvoie `True` si et seulement si n est un nombre premier.

MODULO

Dans la fonction `premier`, il faut vérifier que `n` n'est divisible par aucun des nombres entiers à partir de 2 et strictement inférieurs à lui-même. Pour cela, notons que l'opérateur modulo peut être utile : le test `n % i == 0` est vrai si `i` divise `n` entièrement.

NB : Il est toutefois intéressant de constater qu'il n'est en fait nécessaire de vérifier cette condition que pour les nombres entiers qui sont inférieurs ou égaux à la racine carrée du nombre `n`. Si vous souhaitez rendre votre code plus efficace en réduisant le nombre d'instructions exécutées, la fonction `sqrt` du module `math` s'avèrera utile ici.

ÉNONCÉ EXERCICE UPYLAB 4.3

EXERCICE UPYLAB 4.3 (Vert)

Énoncé

Écrire une fonction booléenne `premier(n)` qui reçoit un nombre entier positif `n` et qui renvoie `True` si `n` est un nombre premier, et `False` sinon.

Ensuite, **écrire un programme** qui lit une valeur entière `x` et affiche, grâce à des appels à la fonction `premier`, tous les nombres premiers **strictement inférieurs** à `x`, chacun sur sa propre ligne.

Exemple 1

Avec la donnée lue suivante :

```
7
```

le résultat à imprimer vaudra donc

```
2
3
5
```

Exemple 2

Avec la donnée lue suivante :

```
9
```

le résultat à imprimer vaudra donc

```
2
3
5
7
```

Consignes

— Dans cet exercice, il vous est demandé d'écrire une fonction, puis un programme appelant cette fonction. Notez qu'UpyLaB testera ces deux points, en exécutant le programme entier mais aussi en appelant directement la fonction avec les arguments de son choix.
— Plus précisément UpyLaB testera d'abord l'existence d'une définition de la fonction `premier` avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera un programme qui réalise 2 appels à la fonction et réalise deux `print` ; ceci pour vérifier que celle-ci n'effectue aucun `print`. C'est en effet à votre programme d'effectuer les `print` demandés. Ensuite différents tests de votre programme complet et de la fonction seront effectués par UpyLaB.
— Il n'est pas demandé que la fonction `premier` teste le type du paramètre reçu, ni si sa valeur est bien positive ou nulle.
— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer, ou pour les fonctions ne renvoyer, que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte dans ce qui est imprimé (`print(res)` et non `print("résultat :", res)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Pour tester si un nombre est premier, l'idée est de chercher, parmi les nombres qui lui sont inférieurs (0 et 1 exclus), un éventuel diviseur. Une instruction itérative, une instruction conditionnelle `if` et l'opérateur `%` (modulo) s'avèreront donc utiles.
— Pour déterminer si le paramètre `n` reçu est premier, votre fonction `premier` doit donc s'assurer que `n` n'est divisible par aucune valeur entière entre 2 et `n - 1` compris.
— Pour le programme principal, ne pas hésiter à s'inspirer de ce qui est proposé dans le texte accompagnant cet exercice.
— Attention, le programme ne doit pas afficher la valeur de *x*, même si celui-ci est premier.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.3*.

4.4 Quelques règles de bonnes pratiques

4.4.1 Quiz "Testez vos connaissances !"

Note : Voir le quiz de la section 4.4.1 du cours en ligne

4.4.2 Code ou programme « propre »

QU'EST-CE QU'UN CODE « PROPRE »

Jusqu'à présent nous avons appris de nouveaux éléments montrant ce qui était possible avec le langage Python. Parfois, il faut aussi parler de ce qui est possible mais doit être évité. La parallèle peut être fait avec l'apprentissage de la vie en société. C'est bien de savoir que l'on peut bouger, communiquer, et satisfaire ses besoins divers, encore faut-il connaître le code de la bonne conduite en société pour ne pas avoir des soucis avec les autres ou se faire ostraciser !

De même, quand un programmeur écrit un code pour résoudre un problème, il faut d'abord que ce code soit **correct**, c'est-à-dire résolve le problème pour toutes les configurations possibles (données, contextes,...). Ensuite il vaut mieux que le code soit **efficace** en prenant le moins de temps possible et en occupant le moins de place mémoire lors de son exécution.

Pour bien programmer, un autre critère essentiel est que le programme ou le code soit **« propre »** : **un code est propre** s'il a un certain nombre de qualités de lisibilité pour un programmeur, de structuration, s'il est bien commenté, ... Python édicte des règles à suivre au mieux lors de l'écriture du code pour que ce dernier soit propre. Libre à chaque programmeur de suivre ces règles ou pas. Mais dans ce dernier cas, ne vous étonnez pas que personne ne communique avec vous vu les difficultés pour les autres de comprendre votre code !

Plus globalement, les recommandations et améliorations pour le langage Python sont répertoriées dans des documents appelés PEPs (Python Enhancement Proposals) sur le site officiel https://www.python.org. Chaque PEP porte un numéro. Le document intitulé « PEP 8 – Style Guide for Python Code » contient un certain nombre de ces **« règles de bonnes pratiques »**. Ne pas suivre ces règles peut aboutir à un programme correct et efficace, mais probablement moins propre. Ces règles parlent de constantes globales, de convention de codage, de taille des fonctions, de conventions typographiques (endroits où l'on met des espaces ou des lignes blanches), ... Nous en parlons dans ce module et le suivant, mais donnons directement un exemple de code qui suit ces règles.

Constantes globales

Nous avons vu qu'un programme complet pouvait contenir des définitions de fonctions, du code, des commentaires, mais également des importations de fonctions prédéfinies ou de modules complets. Généralement, un programme utilise également des valeurs constantes, plus simplement appelées constantes dans le jargon informatique. Ces constantes peuvent être un « paramètre » du programme, par exemple le nombre maximum d'éléments à traiter ou une valeur approchée de pi (3.14159...). Reprenons le petit jeu proposé dans le module précédent en section 3.3, pour améliorer le code proposé. Si notre programme doit deviner un nombre entier entre 0 et 5 que nous avons choisi en utilisant la fonction `randint` du module `random`, il peut être intéressant de donner un nom aux valeurs minimales et maximales de l'intervalle, et donc de définir des « constantes globales » :

```
VALEUR_MIN = 0      # borne inférieure de l'intervalle
VALEUR_MAX = 5      # borne supérieure de l'intervalle
```

Ces constantes seront bien sûr utilisées ensuite tout au long du programme. Un des gros avantages de cette pratique est que si plus tard, le programmeur veut changer cet intervalle de valeurs, il lui suffit de corriger ces deux lignes en début du code.

Structure d'un programme

Les règles de bonnes pratiques nous demandent qu'un programme complet soit formé dans l'ordre :

- du docstring initial du programme contenant, le nom de l'auteur, la date d'écriture, une brève explication de ce qu'il fait, des entrées lues et résultats affichés ;
- des éventuels imports de modules ou de parties de modules ;
- des définitions des constantes globales ;
- des définitions des fonctions globales ;
- et enfin du code du traitement principal.

La découpe du code en fonctions est également importante pour sa clarté. Donnons-en un simple exemple.

Code amélioré du petit jeu de devinette : Continuons avec l'exemple du petit jeu proposé dans le module précédent (section 3.3). Le code suivant vous montre un exemple complet où nous supposons que :

- la valeur aléatoire est choisie dans l'intervalle [VALEUR_MIN, VALEUR_MAX] ;
- l'utilisateur propose bien une valeur entière, mais peut se tromper et proposer une valeur en dehors de l'intervalle.

Par ailleurs, par souci de clarté, nous avons découpé le code grâce à des fonctions :

- une fonction tirage qui s'occupe du tirage aléatoire ;
- une fonction entree_utilisateur qui demande à l'utilisateur son choix et redemande si ce dernier ne propose pas un nombre dans l'intervalle ;
- une fonction affichage_resultat qui affiche à l'écran si l'utilisateur a ou non trouvé le nombre tiré aléatoirement.

L'exemple qui suit respecte bien l'ordre dans lequel les différents éléments de code doivent être placés : import, définitions de constantes globales, de fonctions et finalement code principal.

Nous noterons qu'il respecte également les autres règles prescrites dont nous parlerons plus loin.

```
"""
Petit jeu de devinette (version 2)
Auteur: Thierry Massart
Date : 10 octobre 2018
Petit jeu de devinette d'un nombre entier tiré aléatoirement
par le programme dans un intervalle donné
```

(suite sur la page suivante)

(suite de la page précédente)

```
Entrée : le nombre proposé par l'utilisateur
Résultat : affiche si le nombre proposé est celui tiré
           aléatoirement
"""

# importation des modules

import random   # module le tirage des nombres aléatoires

# Définition des constantes globales

VALEUR_MIN = 0  # borne inférieure de l'intervalle
VALEUR_MAX = 5  # borne supérieure de l'intervalle

# Définition des fonctions

def entree_utilisateur(borne_min, borne_max):
    """
    Lecture du nombre entier choisi par l'utilisateur
    dans l'intervalle [borne_min, borne_max]
    Entrées : bornes de l'intervalle
    Résultat : choix de l'utilisateur
    """
    message = "Votre choix de valeur entre {0} et {1} : "
    ok = False   # vrai quand le choix donné est valable
    while not ok: # répétition tant que le choix n'est pas bon
        choix = int(input(message.format(borne_min, borne_max)))
        ok = (borne_min <= choix and choix <= borne_max)
        if not ok: # entrée hors de l'intervalle
            print("Hors de l'intervalle ! Donnez une valeur valide")
    return choix

def tirage(borne_min, borne_max):
    """
    Tirage aléatoire d'un entier dans [borne_min, borne_max]
    """
    return random.randint(borne_min, borne_max)

def affichage_resultat(secret, choix_utilisateur):
    """
    Affiche le résultat
    """
    if secret == choix_utilisateur:
        print("gagné !")
    else:
        print("perdu ! La valeur était", secret)

# Code principal

mon_secret = tirage(VALEUR_MIN, VALEUR_MAX)
choix_util = entree_utilisateur(VALEUR_MIN, VALEUR_MAX)
affichage_resultat(mon_secret, choix_util)
```

Notes sur format() et f-string

format()

Le code utilise la fonction (appelée méthode dans le jargon) `format` pour imprimer du texte lors d'un input (ici pour demander le choix de l'utilisateur). Cette méthode peut servir, lors d'un print, à formater les impressions, par exemple si l'on veut imprimer un float avec un certain nombre de chiffres après le point décimal.

Notons que l'instruction Python `"Votre choix de valeur entre {0} et {1} : ".format(10, 100)` donne le texte où les « paramètres » `{0}` et `{1}` dans le texte ont été remplacés par les valeurs `10` et `100` donnés commme arguments dans la méthode format.

Pour voir son effet plus précis, essayez par exemple, dans une console PyCharm ou de votre environnement Python 3 :

```
print("pi vaut plus ou moins {0:7}".format(3.14))
```

```
print("pi vaut plus ou moins {0:7.4}".format(3.14159265359))
```

```
print("pi vaut plus ou moins {0:07.4}".format(3.14159265359))
```

Si vous désirez avoir une connaissance approfondie de son fonctionnement, reportez-vous au manuel python qui malheureusement est souvent un peu ardu à lire.

Les f-strings

À partir de Python 3.6, la notion de f-string a été introduite pour remplacer avantageusement la méthode `format()`. Voici deux exemples simples d'utilisation d'une f-string. Pour une description complète, nous vous conseillons le PEP-0498 (la source).

Le premier reprend les trois `print` présentés avec la méthode `format` :

```
print(f"pi vaut plus ou moins {3.14:7}")
print(f"pi vaut plus ou moins {3.14159265359:7.2}")
print(f"pi vaut plus ou moins {3.14159265359:07.4}")
```

```
pi vaut plus ou moins    3.14
pi vaut plus ou moins     3.1
pi vaut plus ou moins 003.142
```

Le deuxième utilise des chaines affectées à des variables :

```
prenom = input('Donnez le prénom du personnage : ')
sexe = input('Donnez le sexe du personnage (F/H) : ')
if sexe == 'H':
    pronom = 'il'
else:
    pronom = 'elle'
print(f'{prenom} entre en scène ; {pronom} est le principal personnage de cette fiction.')
```

```
Donnez le prénom du personnage : Alice
Donnez le sexe du personnage (F/H) : F
Alice entre en scène ; elle est le principal personnage de cette fiction.
```

Une *f-string* consiste donc en une chaîne de caractères qui commence par un `f` ou un `F` **avant** le guillemet ou le quote simple et qui contient, en plus de votre texte, des marqueurs `{ }`. Le marqueur va contenir, une expression que Python sait évaluer ainsi que des informations de formattage pour la valeur calculée.

La syntaxe générale est la suivante :

```
f' <texte> { <expression/variable> : <format> } <texte> ... '
```

Pour des nombres à virgule, le format est sous cette forme (les crochets signifient que les indications sont optionnelles) :

```
[alignement][signe][largeur][groupage][+precision][type]
```

- **alignement** [détermine l'alignement du nombre]
  - > aligne à droite (valeur par défaut)
  - < aligne à gauche
  - ^ centre
  - = aligne le signe à gauche et le nombre à droite
- **signe** [détermine l'affichage du signe]
  - − seul le signe − est affiché (valeur par défaut)
  - + les signes + et − sont affichés
- largeur : détermine le nombre de caractères de la chaîne finale
- **groupage** [détermine le symbole de séparation des milliers (par défaut il n'y a pas de séparation)]
  - _
  - ,
- **type** [détermine le mode d'affichage du flottant]
  - `e` ou `E` pour une notation scientifique
  - `f` ou `F` pour une notation classique

```
print('\ndes flottants')
print(f'{3.14128: f}')
print(f'{-0.7888:.2f}')

print('\ndes entiers')
print(f'{123:=+10}')
print(f'{12998788:=+10}')
```

```
des flottants
 3.141280
-0.79

des entiers
+      123
+ 12998788
```

4.4.3 Quelques éléments supplémentaires

ENCORE QUELQUES CONVENTIONS DE CODAGE

Rajoutons quelques règles de bonnes pratiques pour avoir un code « propre ». Nous avons déjà vu que parmi celles-ci :

- une convention habituelle en Python pour choisir un nom à une variable ou à une fonction est que ce nom soit formé de lettres alphabétiques en minuscules, de chiffres ou du caractère souligné `'_'` et qu'elle évoque le contenu qu'elle représente. Par exemple, `resultat_final` est un nom de variable correct pour recevoir un résultat ;
- si on définit une constante, c'est-à-dire que l'on donne un nom à une valeur que l'on ne modifie pas tout au long de l'exécution du programme, la convention pour donner un nom à cette constante est d'utiliser des majuscules et le caractère souligné, par exemple `DIM_MAX = 100` définit la constante `DIM_MAX` (on suppose que le programmeur ne réassigne pas une autre valeur à `DIM_MAX` ailleurs dans le programme).

En matière de bonnes pratiques, voici encore quelques éléments indispensables :

Indentation : La règle simple est d'ajouter 4 caractères pour chaque nouvelle indentation plutôt que d'utiliser la tabulation. Un bon éditeur de script, par exemple celui fourni dans PyCharm, vous aidera grandement pour cela en faisant la plupart du travail automatiquement.

Lignes de continuation : Les lignes doivent normalement avoir moins de 79 caractères (72 pour les Docstrings). Les blocs de texte plus longs s'écrivent sur plusieurs lignes. Il se peut que le caractère de continuation anti-slash (en anglais backslash) `'\'` en fin de ligne soit nécessaire, même si la plupart du temps, ce caractère de continuation peut être évité quand l'interpréteur sait que l'instruction ou l'expression sur la ligne a une suite.

Par exemple, ayant dans une console Python :

```
>>> ma_variable_dont_le_nom_est_trop_long = 0
```

si nous voulons incrémenter cette variable, la ligne suivante donnera une erreur de syntaxe :

```
>>> ma_variable_dont_le_nom_est_trop_long = ma_variable_dont_le_nom_est_trop_long +
... 1
File "<ipython-input-3-be9f0e911bbc>", line 1
ma_variable_dont_le_nom_est_trop_long = ma_variable_dont_le_nom_est_trop_long +
                                                                               ^
SyntaxError: invalid syntax
```

Une solution simple est d'ajouter des parenthèses à l'expression à droite :

```
>>> ma_variable_dont_le_nom_est_trop_long = (ma_variable_dont_le_nom_est_trop_long +
...                                         1)
```

ou le caractère de continuation :

```
>>> ma_variable_dont_le_nom_est_trop_long = ma_variable_dont_le_nom_est_trop_long + \
...  1
```

ou plus simplement utiliser l'opérateur d'incrémentation :

```
>>> ma_variable_dont_le_nom_est_trop_long += 1
```

Lignes blanches : Les règles de bonnes pratiques conseillent de mettre deux lignes blanches avant et entre chacune des définitions de fonction globales et une ligne blanche pour mettre en évidence une nouvelle partie dans une fonction.

DÉFINITION DE FONCTIONS IMBRIQUÉES

Une dernière remarque dans cette activité : en Python, une fonction `fun_2` peut être définie à l'intérieur d'une autre fonction `fun_1` à condition que `fun_2` soit locale à `fun_1`, c'est-à-dire utilisée uniquement lors du traitement de `fun_1`. Par exemple, dans la fonction `pgcd(x, y)`, on pourrait définir la fonction `suivant` (même si dans ce cas, cela ne simplifie pas fort le code) comme suit :

```
def pgcd(x, y):
    """ calcule le plus grand commun diviseur de deux entier positifs """

    def suivant(x, y):
        """ calcule le couple de valeurs suivantes dans pgcd """
        return (y, x%y)

    # code de pgcd
    while y > 0:
        x, y = suivant (x, y)
    return x
```

La fonction `suivant` n'est utilisée que par `pgcd` : en dehors de l'exécution de `pgcd`, `suivant` n'existe pas. Notons que certains langages de programmation connus, tels que les langages C, C++, Java, ne supportent pas les définitions de fonctions imbriquées.

4.4.4 Petit manuel des règles de bonnes pratiques

PETIT MANUEL À AVOIR À PORTÉE DE CLAVIER

Vous n'avez évidemment pas vu toutes les notions utiles pour écrire un programme Python. Et probablement vous n'avez pas intégré toutes les règles que nous venons de présenter. Ce n'est pas grave ! Pour vous aider, voici un petit manuel que vous pouvez télécharger en cliquant sur le lien petit manuel des bonnes pratiques de programmation (Python) <https://www.fun-mooc.fr/asset-v1:ulb+44013+session05+type@asset+block@BonnesPratiquesPython.pdf, qui donne les règles de bonnes pratiques que nous voudrions que vous respectiez pour coder proprement vos programmes Python.

Pourquoi ne pas imprimer cette page pour toujours l'avoir sous la main quand vous écrirez du code ?

Ce manuel d'une page est découpé en cinq parties :
- traduction d'un problème en programme ;
- programmation ;
- nommage de variables, fonctions, etc. ;
- documentation du code ;
- structure globale d'un programme.

Chaque partie de ce manuel est essentielle et doit être suivie pour veiller à obtenir le code le plus lisible possible.
- La traduction d'un problème en programme passe par une phase d'analyse et est suivie de codage où il faut choisir les fonctions que nous désirons définir et utiliser.
- La partie programmation parle de style de programmation et de quelques erreurs classiques à éviter.
- Nous avons déjà parlé de conventions de nommage des variables, des fonctions, etc.
- La documentation du code signifie que, dans vos programmes, vous devez mettre des commentaires (docstrings ou commentaires en fin de ligne, ...) pour que, si quelqu'un ou même vous, plus tard, relisez le code, il sera facile de comprendre ce qu'il fait. La documentation est donc inutile pour l'ordinateur, et en particulier pour l'interpréteur, mais est bien utile pour les pauvres humains que nous sommes qui essayons de comprendre ce que fait un code !
- Enfin, la dernière partie rappelle et illustre la structure globale d'un programme.

Ce petit manuel est également donné ci-dessous : à vous de lire et d'apprendre l'ensemble des consignes !

Une dernière bonne nouvelle : l'éditeur de script PyCharm intègre la plupart de ces règles et vous indique là où il considère que vous ne les respectez pas.

LE MANUEL DE BONNES PRATIQUES

Le petit manuel des bonnes pratiques Python est accessible en https://www.fun-mooc.fr/asset-v1:ulb+44013+session05+type@asset+block@BonnesPratiquesPython.pdf

4.4.5 La règle du return unique et les instructions dont on ne veut pas prononcer le nom

RÈGLES POUR NE PAS ENTRER PAR LA PORTE ET SORTIR PAR LA FENÊTRE

Pour ceux, comme vous probablement, qui débutent l'apprentissage de la programmation, nous avons l'habitude de rajouter quelques « règles de bonnes pratiques » supplémentaires. Certaines de ces règles peuvent être assouplies pour les programmeurs avertis qui n'ont plus besoin de « guide » pour bien coder.

Ces règles obligent les codes à respecter l'ordre « normal » d'exécution. Par exemple, si le code contient une instruction « if », « while » ou « for » ou une séquence d'instructions, il nous semble important pour bien comprendre la logique du code, de ne pas interrompre ce code en plein milieu.

Malheureusement la plupart des langages de programmation dits « impératifs » comme Python permet de déroger à cette règle en offrant au programmeur certaines instructions de « rupture de séquence ». Nous ne vous en avons pas parlé puisque nous ne voulons pas que vous les utilisiez ! *Nous comparons souvent l'utilisation de ce type d'instructions, à quelqu'un qui visite une maison et qui en sort par la fenêtre : ce n'est pas très poli !*

En fait au sein des fonctions, nous vous avons parlé d'une de ces instructions : l'instruction `return`. Elle permet de sortir d'une instance de fonction pour revenir au code appelant. Le fonctionnement de Python oblige de terminer l'exécution de toute fonction

par un `return` explicite si l'on veut renvoyer une valeur précise, ou implicite si la fonction renvoie la valeur None. Il n'est donc pas question de proscrire l'instruction `return`, mais bien de la cadrer de façon précise.

La règle suivante est généralement une bonne pratique :

— Une fonction de type `None` ne doit pas contenir de `return`.
— Une fonction d'un type différent de `None` ne doit contenir qu'un seul `return` avec le résultat, et ce `return` doit être placé comme dernière instruction de cette fonction en dehors de toute instruction composée (if, while, for, ...).

Habituez-vous à respecter cette règle qui vous permet d'écrire du code plus « propre » : ce n'est pas si difficile !

Et pourquoi ne pas reprendre vos codes UpyLaB précédents et s'ils ne suivent pas les règles de bonnes pratiques de codage, les corriger ?

Exception

Cette règle peut être assouplie quand le code de la fonction produit avec plusieurs return est beaucoup plus simple qu'avec un seul.

Un canevas « classique » est celui d'une recherche dans une séquence d'éléments avec arrêt quand on a trouvé ce que l'on cherchait. Par exemple si l'on désire écrire une fonction booléenne *divisible_2_3_5_7* qui reçoit en paramètre un entier *borne_max* et qui renvoie *True* si un nombre entre 1 et *borne_max* non compris est divisible par 2, 3, 5 et 7, un code simple peut être :

```
def divisible_2_3_5_7(borne_max):
    for i in range(1, borne_max):
        if i %2 == 0 and i % 3 == 0 and i % 5 == 0 and i % 7 == 0:
            return True
    return False
```

La solution avec un seul return en fin donnerait par exemple :

```
def divisible_2_3_5_7(borne_max):
    res = False
    i = 1
    while  not res and i < borne_max:
        res =  i %2 == 0 and i % 3 == 0 and i % 5 == 0 and i % 7 == 0
        i += 1
    return res
```

Notons qu'un code plus efficace qui n'analyse que les nombres pairs est possible mais ce n'est pas le propos ici. Par contre le premier code montre un parcours d'une séquence éventuellement interrompu, si on a trouvé ce que l'on cherche ; le second, qui se soucie d'avoir un seul *return*, est plus compliqué.

LE PEP 20 EN MODE MONTY PYTHON (PARTIE OPTIONNELLE)

Parmi les recommandations et améliorations pour le langage Python, répertoriées dans les PEPs (Python Enhancement Proposals), nous trouvons le PEP 20. Le PEP 20 résume les 20 aphorismes (dont seulement 19 ont été rédigés) utilisés par Guido van Rossum, le BDFL Python (Benevolent Dictator for Life) jusqu'en 2018, pour concevoir le langage lui-même.

(Note : Le dictionnaire Larousse définit aphorisme par : phrase, sentence qui résume en quelques mots une vérité fondamentale. (Exemple : Rien n'est beau que le vrai.))

La lecture du PEP 20 est assez hermétique pour les programmeurs débutants ; ne vous arrêtez donc pas à cela. Si vous êtes quand même curieux de son contenu (en anglais) ouvrez une console PyCharm et tapez la commande :

```
import this
```

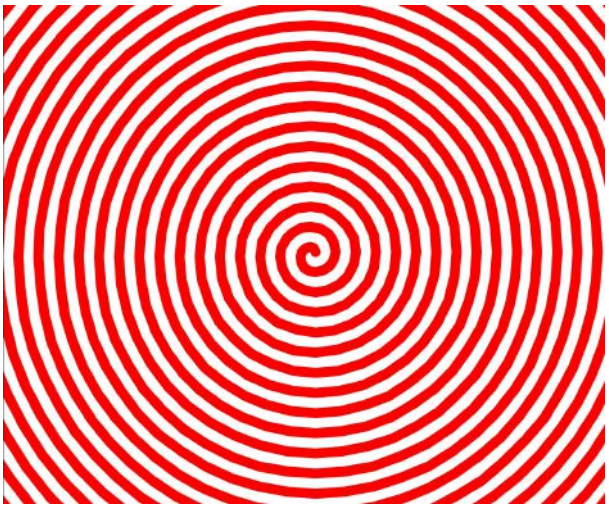
qui vous dévoilera son contenu inspiré par l'humour absurde des Monty Python dont Guido van Rossum était féru.

4.5 Pratique des fonctions

4.5.1 Mettons ensemble en pratique ce que nous venons de voir

SPIRALE AVEC TURTLE

Comme exercice, nous vous demandons de dessiner une spirale ou une courbe assez proche d'une spirale avec le module turtle. La technique que nous vous proposons pour dessiner la spirale est assez simple : il suffit d'assembler bout à bout des quart de cercles de rayon de plus en plus grand.



La vidéo suivante montre l'affichage d'un petit programme qui appelle une fonction `spirale`. Le code utilise turtle et la fonction `circle` pour dessiner des quarts de tour de plus en plus grands. Les autres fonctions du module turtle utilisées par mon code sont `reset`, `color`, `speed` et `width`.

VIDÉO DE LA SPIRALE AVEC TURTLE

Note : Voir la vidéo de la section 4.5.1 : Exemple de dessin de spirale

EXERCICE À FAIRE ENSEMBLE

L'exercice suivant met en pratique ce que nous avons vu sur les fonctions. Écrivez, dans un script Python, un code qui réalise un affichage similaire grâce à une fonction `spirale` qui a deux paramètres :

— la couleur,
— la largeur du trait.

Ainsi, la figure précédente est l'exécution répétée de : `spirale('red', 10)`

PROPOSITION DE SOLUTION

Vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir un autre code qui solutionne l'exercice demandé ? Nous vous proposons une solution.

```
""" auteur: Thierry Massart
    date : 10 avril 2018
    but du programme : trace avec turtle des spirales
"""

import turtle

def spirale(couleur, largeur):
    """ Dessine avec turtle une spirale
    paramètres :
    - couleur : la couleur à utiliser
    - largeur : la largeur du tracé"""

    turtle.color(couleur)
    turtle.width(largeur)
    for i in range(100):
        turtle.circle(i*(largeur/2),90)

while True:
    turtle.reset()
    turtle.speed(0)
    spirale('red', 10)
```

Notez que la solution proposée boucle indéfiniment : le programme n'arrêtera son exécution que lorsque l'on le « tue », par exemple avec la touche clavier cmd-F2 (commande-F2) ou CTRL-c (contrôle-c) en étant dans la fenêtre Run du script PyCharm en exécution.

4.5.2 Autre exercice accompagné

POLYGONES AVEC TURTLE

Continuons de pratiquer la matière de ce module avec turtle qui nous permet de facilement visualiser le résultat et donc d'en valider notre compréhension. Supposons que nous désirions dessiner diverses figures. Nous pouvons, pour ce faire, imaginer définir différentes fonctions. Prenons le cas d'une fonction qui dessine un polygone régulier.

À nouveau, la première chose à déterminer est le nom de la fonction et quels paramètres elle aura.

Supposons que l'on appelle cette fonction `polygone_turtle`. Il faut ensuite penser à l'ensemble des paramètres.

Les paramètres pourraient être :

— le nombre de côtés du polygone régulier ;
— les coordonnées (x, y) du centre de la figure sur la fenêtre turtle ;
— la taille du rayon du cercle qui circonscrit le polygone ;
— la couleur du polygone.

Par exemple, le dessin suivant provient d'un programme qui a appelé la fonction `polygone_turtle` et défini cinq couleurs différentes, le nombre de côtés, le centre et le rayon étant choisis aléatoirement.

RÉSULTAT DU PROGRAMME POLYGONE_TURTLE.PY

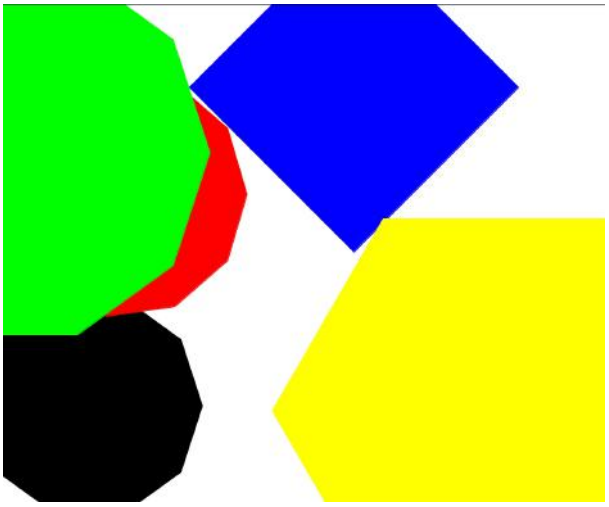


Fig. 4.4 – Exemple de dessin de polygônes

Le code dessine séquentiellement 5 polygones remplis : noir, bleu, rouge, vert et jaune, de taille, nombre de côtés et emplacement choisis pseudo-aléatoirement.

EXERCICE À RÉALISER

À vous !

Écrivez dans un script Python un programme complet `figures_geometriques` qui définit et utilise cette fonction `polygone_turtle`. Vous pouvez également le compléter d'une fonction `etoile_turtle(n, x, y, rayon, couleur)` qui dessine une étoile à n branches avec les mêmes spécifications qu'avec la fonction `polygone_turtle`.

Ici nous ne vous donnons pas une solution complète. Comme le résultat est graphique, il vous sera possible de vérifier s'il fonctionne correctement. Nous vous donnons quand même une aide pour réaliser le programme `figures_geometriques`, si vous avez des soucis avec les formules trigonométriques.

Petite aide pour réaliser le programme figures_geometriques

Ayant importé `turtle`, `cos`, `sin` et `pi`, après avoir positionné, avec `turtle.goto(x+rayon, y)` la tortue au premier point du polygone de centre `(x, y)` et de rayon `rayon`, le code suivant trace le polygone à n côtés

```
for i in range(1, n + 1):
  turtle.goto(x + rayon * cos(i * 2 * pi / n), y + rayon * sin(i * 2 * pi / n))
```

4.5.3 Le yin et le yang revisité et paramètres par défaut

YIN-YANG REVISITÉ

Reprenons le code de la fonction `yin_yang` donné en section 4.3 pour le corriger en suivant les bonnes pratiques discutées avant dans ce module. Cette fonction avait été proposée sans paramètre. Pour respecter les règles de bonnes pratiques d'un codeur Python, cette fonction a plusieurs défauts :

1. elle est trop longue en terme de séquence monolithique de lignes. Il faudrait essayer de la découper en utilisant plusieurs sous-fonctions.
2. elle devrait avoir des paramètres :

— ici le yin-yang a un rayon de 200 points. Si nous voulons faire un yin-yang d'une autre taille, il faudrait corriger des valeurs un peu partout dans le code, ce qui peut être une belle source d'erreurs.
— on pourrait imaginer un yin-yang avec d'autres couleurs, non centré, ...

3. si l'on regarde le code, il fait plus ou moins deux fois la même chose puisque le yin à la même forme que le yang. Seuls l'emplacement et la couleur changent. Il serait donc bien d'essayer de n'avoir qu'un seul code qui trace à la fois le yin et le yang en fonction des arguments donnés à l'exécution.

PARAMÈTRES AVEC VALEUR PAR DÉFAUT

Python propose de donner des valeurs par défaut aux paramètres lors de sa définition. Rappelons que « par défaut » signifie « si rien n'est donné explicitement ». Ici, « par défaut » signifie donc « si lors de l'appel à la fonction la valeur de l'argument n'est pas donnée ».

Par exemple nous pouvons définir l'entête d'une fonction `yin_yang` comme suit : `def yin_yang(rayon, color1='black', color2='white'):` le corps de la fonction étant défini normalement. Avec cet entête, si lors de l'appel à la fonction `yin_yang`, seulement deux paramètres sont donnés, le premier argument sera donné pour le paramètre `rayon` et le second pour le paramètre `color1`. Le paramètre `color2` prendra la valeur par défaut `'white'`.

Si lors d'un appel à la fonction `yin_yang`, un seul argument est donné (pour le paramètre `rayon`), `color1` vaudra sa valeur par défaut `'black'` et `color2` sa valeur par défaut c'est-à-dire `'white'`. Notons que comme le paramètre `rayon` n'a pas de valeur par défaut dans sa définition, il faudra au moins donner un argument lors de tout appel à la fonction.

Notons que Python a également la possibilité d'effectuer la transmission des arguments aux paramètres par mot-clé comme avec `print(res1, res2, res3, end="", sep="/")` qui spécifie que les paramètres `end` et `sep` prennent respectivement la valeur chaîne de caractères vide (`""`) et la caractère barre oblique.

À vous de jouer !

Nous vous proposons donc, à nouveau dans votre IDE, de modifier le code précédent pour mieux le structurer et permettre d'autres couleurs (le contour restant en noir).

Pour vous aider, voici la structure proposée : la fonction `yin_yang` définit en son sein deux fonctions :
— la fonction `yin` qui fait la moitié du symbole (soit le yin soit le yang). Elle sera donc appelée deux fois par la fonction `yin_yang`;
— la fonction `yang` qui fait le disque yang (respectivement yin) à l'intérieur du yin (resp. yang) et qui est appelée au sein de la fonction `yin`.
L'ossature du code proposé est donc comme suit :

OSSATURE CODE PYTHON

```
""" code avec fonction yin_yang dessiné avec turtle """
import turtle

def yin_yang(rayon, color1='black', color2='white'):
    """ dessine un logo yin-yang de rayon rayon """

    def yang (rayon, couleur1, couleur2):
        """ dessin du yang à l'intérieur du yin (ou vice versa) """
        pass # code TODO

    def yin(rayon, couleur1, couleur2):
        """ dessine la moitié d'un yin-yang
            utilise la fonction yang """
        pass # code TODO

    #code de yin_yang
    turtle.reset()
```

(suite sur la page suivante)

(suite de la page précédente)

```
    turtle.width(2)
    yin(rayon, color1, color2)
    yin(rayon, color2, color1)
    turtle.hideturtle()
    return

#code principal
yin_yang(200) #réalise le logo de rayon 200
```

Si vous ne trouvez pas la bonne réponse, n'hésitez pas à en discuter via le [Forum général du Module 4] pour y arriver !

Si malgré tout, vous avez du mal pour réaliser l'exercice ou vous l'avez réussi mais voulez avoir une autre solution, nous vous en proposons une.

PROPOSITION DE SOLUTION

Ci-dessous une proposition de solution.

```
""" auteur : Thierry Massart
    date : 10 avril 2018
    code avec fonction yin_yang dessiné avec turtle """
import turtle

def yin_yang(rayon, color1='black', color2='white'):
    """ dessine un logo yin-yang de rayon rayon """

    def yang (rayon, couleur1, couleur2):
        """ dessin du yang à l'intérieur du yin """
        turtle.left(90)
        turtle.up()
        turtle.forward(rayon*0.35)
        turtle.right(90)
        turtle.down()
        turtle.color(couleur1, couleur2)
        turtle.begin_fill()
        turtle.circle(rayon*0.12)
        turtle.end_fill()
        turtle.left(90)
        turtle.up()
        turtle.backward(rayon*0.35)
        turtle.down()
        turtle.left(90)

    def yin(rayon, couleur1, couleur2):
        """ dessine la moitié d'un yin-yang le contour étant en noir"""
        turtle.color("black", couleur1)
        turtle.begin_fill()
        turtle.circle(rayon/2., 180)     # demi cercle de rayon / 2
        turtle.circle(rayon, 180)        # demi cercle de rayon
        turtle.left(180)
        turtle.circle(-rayon/2., 180)    # demi cercle intérieur de rayon / 2
        turtle.end_fill()
        yang(rayon, couleur1, couleur2) # dessine le yang à l'intérieur du yin

    #code de yin_yang
    turtle.reset()
    turtle.width(2)
    yin(rayon, color1, color2)
```

(suite sur la page suivante)

(suite de la page précédente)

```
    yin(rayon, color2, color1)
    turtle.hideturtle()
    return

#code principal
yin_yang(200) #réalise le logo
```

4.5.4 Un pavé hexagonal avec fonctions

ÉNONCÉ DE L'EXERCICE

Nous avons écrit durant l'activité 2.6.3, du code pour tracer un pavé. L'activité suivante réalise le codage d'une fonction :

```
pave(abscisse_centre, ordonnee_centre, longueur_arete, color1, color2, color3)
```

qui trace dans une fenêtre turtle

- à partir du point centre donné par les coordonnées `(abscisse_centre, ordonnee_centre)`
- un pavé dont chaque arête à une longueur `longueur_arete`
- avec des 3 losanges remplis respectivement par les `color1`, `color2` et `color3`.

Pavé noir, bleu et rouge

Votre fonction déplacera d'abord turtle au point centre sans tracer (grâce à la fonction `turtle.up`) et ensuite tracera le pavé (fonction `turtle.down` et `turtle.goto`).

Conseils :

- Comme lors du composant 2.6.3, nous vous conseillons d'utiliser la fonction `goto` pour effectuer les tracés. Attention ici, si le pavé n'est pas au centre de la fenêtre, il faut tenir compte des coordonnées du centre lors des appels à `turtle.goto`.
- Pour simplifier, utilisez une version sans instruction `for` ou `while`.

Vous pourrez ensuite tester votre fonction `pave` avec un code l'appelant avec différentes valeurs d'arguments pour dessiner des pavés à différents endroits de la fenêtre turtle et de différentes tailles.

Par exemple vous pouvez compléter le code suivant, qui fait différents appels avec des valeurs aléatoires, en ajoutant votre code de la fonction `pave`.

CANEVAS DE CODE

```
import turtle
from math import pi, sin, cos
import random
import time

def pave(abscisse_centre, ordonnee_centre, longueur_arete, color1, color2, color3):
    """ Dessine avec turtle un pave hexagonal
        en position ( abscisse_centre, ordonnee_centre)
        paramètres :
        - (abscisse_centre, ordonnee_centre) : point centre du pavé
        -  longueur_arete : longueur de chaque arête du pavé
        - color1, color2, color3 : les couleurs des 3 hexagones"""
    # TODO code de fonction pave ici

turtle.hideturtle()
turtle.speed(0)
turtle.reset()
time.sleep(5)
```

(suite sur la page suivante)

(suite de la page précédente)

```
while True:
    pave(random.randint(-300,300),  random.randint(-300,300),
        random.randint(10,50), 'black', 'red', 'blue')
    pave(random.randint(-300,300),  random.randint(-300,300),
        random.randint(10,50), 'white', 'grey', 'black')
```

Votre code complet devrait produire un résultat similaire à celui de la vidéo de la section 4.5.5

RÉSULTAT DU CODE

Note : Voir la vidéo de la section 4.5.5 : Dessins de pavés hexagonaux

Note : Notez que la seconde partie de la vidéo est la séquence vidéo inversée réalisée avec le logiciel screenflow.

4.5.5 Mise en pratique : exercice UpyLaB 4.4 et sq

MISE EN PRATIQUE AVEC UPYLAB

Nous sommes arrivés à la fin de ce module, où vous devez devenir autonome dans la rédaction de code Python utilisant des fonctions. Pour cela réalisez tous les exercices UpyLaB du module 4 qu'il vous reste à faire.

N'oubliez pas qu'UpyLaB est un environnement de test et non un environnement de développement. En particulier, la plupart des codes demandés par la suite par UpyLaB ne sont qu'une partie d'un programme complet. Par exemple, UpyLaB demande de définir une fonction `foo`, et complètera votre définition par un code qui testera l'exécution de votre fonction pour plusieurs jeux de paramètres. Avant de soumettre à UpyLaB, il est fortement conseillé de développer dans un environnement de développement (Thonny ou PyCharm par exemple) un programme complet qui définisse `foo` mais également qui fasse des appels pour tester si elle est correcte. C'est ce que l'on appelle réaliser des test unitaires ; l'unité étant un bout de code comme une fonction, ici la fonction `foo`.

Bon travail !

EXERCICE UPYLAB 4.4 (Vert)

Énoncé

Écrire une fonction `alea_dice(s)` qui génère trois nombres (pseudo) aléatoires à l'aide de la fonction `randint` du module `random`, représentant trois dés (à six faces avec les valeurs de 1 à 6), et qui renvoie la valeur booléenne `True` si les dés forment un 421, et la valeur booléenne `False` sinon.

Le paramètre `s` de la fonction est un nombre entier, qui sera passé en argument à la fonction `random.seed` au début du code de la fonction. Cela permettra de générer la même suite de nombres aléatoires à chaque appel de la fonction, et ainsi de pouvoir tester son fonctionnement.

Exemple 1

L'appel suivant de la fonction :

```
alea_dice(1)
```

doit retourner :

```
False
```

Exemple 2

L'appel suivant de la fonction :

```
alea_dice(25)
```

doit retourner :

```
True
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire la fonction `alea_dice`. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Plus précisément UpyLaB testera d'abord l'existence d'une définition de votre fonction avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre fonction : il ajoutera des appels à votre fonction pour vérifier que celle-ci n'effectue aucun `print`. Ensuite différents tests de votre fonction seront effectués par UpyLaB.
— Il n'est pas demandé que la fonction `alea_dice` teste le type du paramètre reçu.
— N'importe quelle combinaison de trois dés permettant de former le nombre 421 sera acceptée, quel que soit l'ordre de la combinaison. Par exemple, les tirages 2, 4, 1 forment bien 421.
— La première instruction de la fonction, après l'import du module `random`, sera `random.seed(s)`.
— On rappelle que la fonction `randint(a, b)` retourne un nombre (pseudo) aléatoire compris entre les bornes `a` et `b` incluses.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Veillez à ce que la fonction retourne bien les valeurs booléennes `True` ou `False` et non les chaînes de caractères `"True"` ou `"False"`.
— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter l'instruction `print(alea_dice(s))`, en remplaçant `s` par une valeur entière. En particulier, vous pouvez remplacer ce paramètre `s` par l'argument utilisé par UpyLaB dans chacun de ces tests.
— Python Tutor pourra s'avérer particulièrement utile ici pour observer ce qu'il se passe exactement lors de l'appel de la fonction.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.4*.

Note pour réaliser les exercices qui suivent

Certains exercices dont l'exercice UpyLaB suivant utilisent des valeurs de type tuple. Les `tuples` sont des séquences de données qui seront étudiées en détail au module suivant. Pour réaliser les exercices de ce présent module, ayant des valeurs `v1` et `v2` (par exemple `int` ou `float`), il faut juste savoir que :

— `tuple()` : renvoie un tuple vide
— `(v1, )` : renvoie un tuple avec une composante de valeur v1
— `(v1, v2)` : renvoie un tuple avec les deux valeurs v1 et v2.
— ...

Ainsi `(2, 3, 5, 7, 11)` est un 5-tuple (contient 5 valeurs).

4.5.6 Exercice UpyLaB 4.5 (Vert)

Énoncé

Considérons les billets et pièces de valeurs suivantes : 20 euros, 10 euros, 5 euros, 2 euros et 1 euro.

Écrire une fonction `rendre_monnaie` qui reçoit en paramètre un entier `prix` et cinq valeurs entières `x20`, `x10`, `x5`, `x2` et `x1`, qui représentent le nombre de billets ou de pièces de chaque valeur que donne un client pour l'achat d'un objet dont le prix est mentionné.

La fonction doit renvoyer cinq valeurs, représentant le nombre de billets et pièces de chaque sorte qu'il faut rendre au client, dans le même ordre que précédemment. Cette décomposition doit être faite en rendant le plus possible de billets et pièces de grosses valeurs.

Si la somme d'argent avancée par le client n'est pas suffisante pour effectuer l'achat, la fonction retournera cinq valeurs `None`.

Exemple 1

L'appel suivant de la fonction :

```
rendre_monnaie(38, 1, 1, 1, 1, 1)
```

doit retourner :

```
(0, 0, 0, 0, 0)
```

Exemple 2

L'appel suivant de la fonction :

```
rendre_monnaie(56, 5, 0, 0, 0, 0)
```

doit retourner :

```
(2, 0, 0, 2, 0)
```

Exemple 3

L'appel suivant de la fonction :

```
rendre_monnaie(80, 2, 2, 2, 3, 3)
```

doit retourner :

```
(None, None, None, None, None)
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Plus précisément UpyLaB testera d'abord l'existence d'une définition de votre fonction avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre fonction : il ajoutera des appels à votre fonction pour vérifier que celle-ci n'effectue aucun `print`. Ensuite différents tests de votre fonction seront effectués par UpyLaB.
— Il n'est pas demandé que la fonction `rendre_monnaie` teste le type des paramètres reçus.
— On suppose qu'il y a toujours suffisamment de billets et pièces de chaque sorte.
— Pour retourner cinq valeurs, on pourra utiliser l'instruction `return res20, res10, res5, res2, res1`.

Cela renvoie en réalité un tuple de cinq valeurs (apparaissant entre parenthèses lorsqu'on l'affiche). La notion de tuple sera introduite au module suivant ; pour l'instant, ne vous préoccupez pas de ceci.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(rendre_monnaie(70, 2, 1, 2, 2, 2))` par exemple.
— Pour déterminer le nombre de billets et pièces de chaque sorte, l'opérateur `//` pourra s'avérer utile, ainsi que l'opérateur `%` pour calculer la somme restant à décomposer.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.5*.

4.5.7 Exercice UpyLaB 4.6 (Vert)

Énoncé

Dans cet exercice, nous allons mettre en pratique la notion de valeur par défaut des paramètres d'une fonction.

Écrire une fonction `somme(a, b)` qui retourne la somme de deux valeurs entières `a` et `b`. Par défaut, la valeur de `a` est 0 et la valeur de `b` est 1.

Exemple 1

L'appel suivant de la fonction :

```
somme(24, 18)
```

doit retourner :

```
42
```

Exemple 2

L'appel suivant de la fonction :

```
somme(4)
```

doit retourner :

```
5
```

Exemple 3

L'appel suivant de la fonction :

```
somme()
```

doit retourner :

```
1
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Plus précisément UpyLaB testera d'abord l'existence d'une définition de votre fonction avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre fonction : il ajoutera des appels à votre fonction pour vérifier que celle-ci n'effectue aucun `print`. Ensuite différents tests de votre fonction seront effectués par UpyLaB.
— Il n'est pas demandé que la fonction `somme` teste le type des paramètres reçus.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(somme(3, 2))` par exemple.
— N'hésitez pas à revoir comment on a défini des valeurs par défaut pour certains paramètres de la fonction `yin_yang`.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.6*.

4.5.8 Exercice UpyLaB 4.7 (Bleu)

Énoncé

Écrire une fonction `rac_eq_2nd_deg(a, b, c)` qui reçoit trois paramètres de type float correspondant aux trois coefficients de l'équation du second degré $ax^2 + bx + c = 0$, où a est différent de 0, et qui renvoie la ou les solutions s'il y en a, sous forme d'un tuple.

Exemple 1

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0,-4.0,4.0)
```

doit retourner :

```
(2.0,)
```

Exemple 2

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0,1.0,-2.0)
```

doit retourner :

```
(-2.0, 1.0)
```

Exemple 3

L'appel suivant de la fonction :

```
rac_eq_2nd_deg(1.0,1.0,1.0)
```

doit retourner :

```
()
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Plus précisément UpyLaB testera d'abord l'existence d'une définition de votre fonction avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre fonction : il ajoutera des appels à votre fonction pour vérifier que celle-ci n'effectue aucun `print`. Ensuite différents tests de votre fonction seront effectués par UpyLaB.
- Il n'est pas demandé que la fonction `rac_eq_2nd_deg` teste le type des paramètres reçus.
- Le résultat retourné par la fonction `rac_eq_2nd_deg` est un tuple.
  - S'il n'y a pas de solution réelle, elle retourne un tuple vide `tuple()`.
  - S'il y a une unique racine `r1`, elle retourne le tuple `(r1,)`.
  - S'il y a deux solutions réelles, `r1` et `r2`, la plus petite des deux devra être la première composante du tuple retourné (composante d'indice 0). La fonction pourra retourner le tuple `(min(r1, r2), max(r1, r2))`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(rac_eq_2nd_deg(1.0, 1.0, 1.0))` par exemple.
— Les calculs permettant d'obtenir les éventuelles solutions de l'équation ont été présentés au module 3 de ce cours, lors de l'introduction de l'instruction conditionnelle `if`.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.7*.

4.5.9 Exercice UpyLaB 4.8 (Bleu)

Énoncé

De Wikipedia (5 février 2019) :

En mathématiques, et plus particulièrement en combinatoire, les **nombres de Catalan** forment une suite d'entiers naturels utilisée dans divers problèmes de dénombrement.

Ils sont nommés ainsi en l'honneur du mathématicien belge Eugène Charles Catalan (1814-1894).

Les dix premiers nombres de Catalan (pour n de 0 à 9) sont :

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862.

Le nombre de Catalan d'indice n, appelé n-ième nombre de Catalan, est défini par :

$$C_n = \frac{(2n)!}{(n+1)!n!}$$

où $n!$ désigne la factorielle de la valeur entière n :

$$n! = n(n-1)(n-2)...1$$

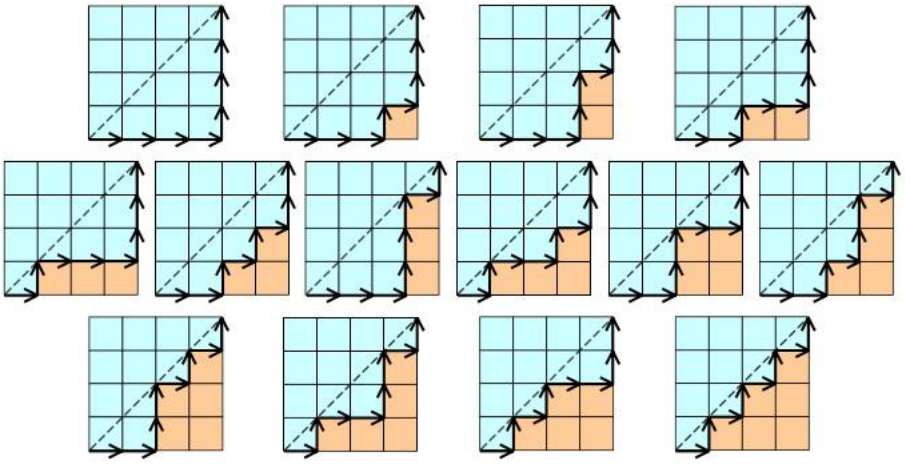
Par exemple, $5! = 5.4.3.2.1 = 120$ et

$$C_4 = \frac{8!}{5!4!} = 14$$

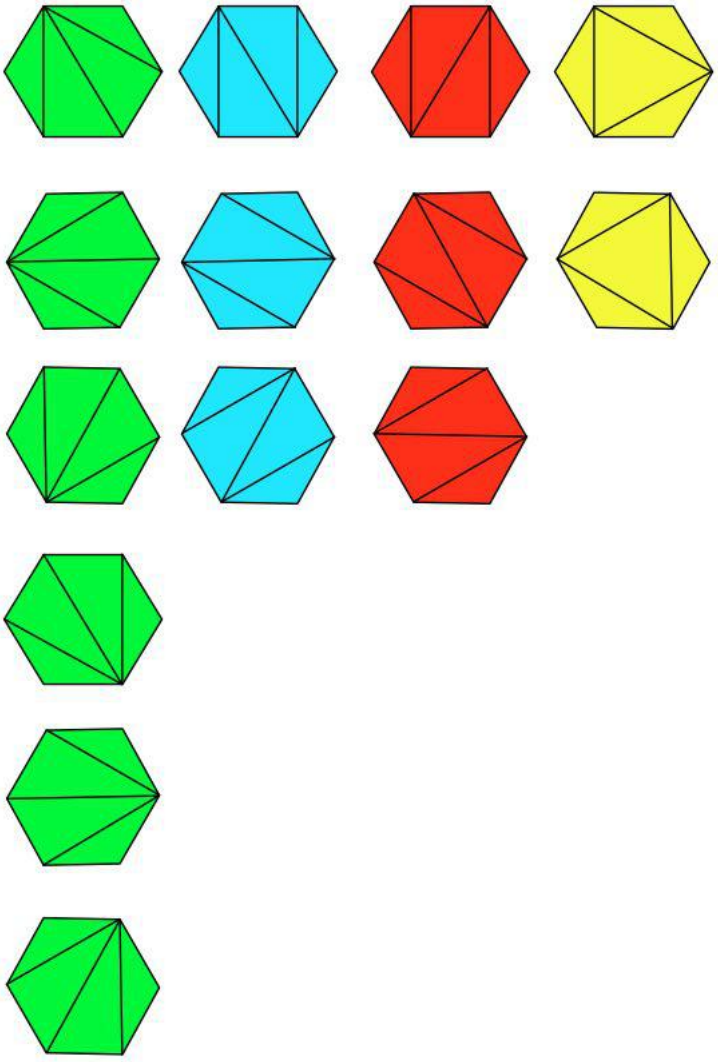
Le nombre de chemins sous-diagonaux les plus courts dans une grille de taille $n \times n$, le nombre de façons de découper en triangles un polygone convexe à $n + 2$ côtés, ou encore le nombre de configurations possibles d'expressions avec n paires de parenthèses, appelé également mot de Dyck de longueur $2n$, sont des exemples dont le résultat est donné par le nombre de Catalan C_n.

Exemples d'applications de C_n (ici, $n = 4$)

— Nombre de chemins sous-diagonaux les plus courts dans une grille de taille $n \times n$



— Nombre de façons de découper en triangles un polygone convexe de taille $n + 2$



— Nombre de parenthésages possibles (mots de Dyck)

((((()))))	(((())))	(((())())	(()(()))	()((()))	(()()())	(((()))()
()(()())	(())(())	(()())()	(())()()	()(())()	()()(())	()()()()

Écrire une fonction `catalan(n)`, où `n` est un nombre entier positif ou nul, qui renvoie la valeur du `n`-ième nombre de Catalan.

Exemple 1

L'appel suivant de la fonction :

```
catalan(5)
```

doit retourner :

```
42
```

Exemple 2

L'appel suivant de la fonction :

```
catalan(0)
```

doit retourner :

```
1
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Il n'est pas demandé que la fonction `catalan` teste le type du paramètre reçu.
- Plus précisément UpyLaB testera d'abord l'existence d'une définition de votre fonction avec le bon nombre de paramètres. Si la fonction existe bien, UpyLaB testera votre fonction : il ajoutera des appels à votre fonction pour vérifier que celle-ci n'effectue aucun `print`. Ensuite différents tests de votre fonction seront effectués par UpyLaB.
- Votre fonction doit renvoyer une valeur **entière**.
- Rappelons aussi que si une fonction est demandée, UpyLaB va tester à la fois que les résultats envoyés par cette fonction sont corrects et qu'ils ont le bon type.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Vous pouvez utiliser la fonction `factorial` du module `math`, ou écrire votre propre fonction `factorielle` ou encore ne pas calculer explicitement ces factorielles en étudiant comment l'on passe d'un nombre de Catalan au suivant.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 4.8*.

4.5.10 Exercice UpyLaB 4.9 (Bleu)

Cet exercice et le suivant vous demandent de programmer le petit jeu appelé « Pierre-feuille-ciseaux » ou « Pierre-papier-ciseaux » (et qui porte encore d'autres noms comme indiqué dans la page Pierre-papier-ciseaux sur Wikipedia que nous utilisons pour rédiger l'énoncé ci-dessous).

Énoncé

Pierre-feuille-ciseaux est un jeu effectué avec les mains et qui oppose un ou plusieurs joueurs. Ici nous nous supposerons qu'il y a deux joueurs : l'ordinateur et le joueur lui-même.

Déroulement du jeu

Les deux joueurs choisissent simultanément un des trois coups possibles en le symbolisant de la main :

- Poing fermé : Pierre ;
- Main ouverte, doigts collés les uns aux autres : Feuille ;
- Main avec pouce, annulaire et auriculaire fermé, index et majeur ouvert en forme de V : Ciseaux.

Résultat du jeu :

- La pierre bat les ciseaux (en les émoussant),
- les ciseaux battent la feuille (en la coupant),
- la feuille bat la pierre (en l'enveloppant).

Ainsi chaque coup bat un autre coup, fait match nul contre le deuxième (son homologue) et est battu par le troisième.

Écrire une fonction `bat(joueur_1, joueur_2)` où `joueur_1` et `joueur_2` ont chacun une valeur entière 0, 1 ou 2, qui encode ce que le joueur a fait comme coup (0 : PIERRE, 1 : FEUILLE, 2 : CISEAUX) qui renvoie un résultat booléen :

— vrai si `joueur_1` bat le `joueur_2` :
— faux si `joueur_2` bat `joueur_1` ou fait match nul contre lui.

Exemple 1

L'appel suivant de la fonction :

```
bat(0, 0)
```

doit retourner :

```
False
```

Exemple 2

L'appel suivant de la fonction :

```
bat(0, 1)
```

doit retourner :

```
False
```

Exemple 3

L'appel suivant de la fonction :

```
bat(2, 1)
```

doit retourner :

```
True
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes et des trois définitions des constantes PIERRE, FEUILLE, CISEAUX, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Il n'est pas demandé que la fonction `bat` teste le type ou les valeurs des paramètres reçus.

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(bat(2, 2))` par exemple. Notez que comme chaque paramètre peut avoir 3 valeurs, la fonction `bat` doit gérer 9 cas possibles. N'hésitez pas à les tester tous avant de mettre votre code dans UpyLaB.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.9*.

4.5.11 Exercice UpyLaB 4.10 (Rouge)

Énoncé

Pierre-feuille-ciseaux est un jeu effectué avec les mains et qui oppose un ou plusieurs joueurs.

Déroulement du jeu

Les deux joueurs choisissent simultanément un des trois coups possibles en le symbolisant de la main :

— Poing fermé : Pierre ;
— Main ouverte, doigts collés les uns aux autres : Feuille ;
— Main avec pouce, annulaire et auriculaire fermé, index et majeur ouvert en forme de V : Ciseaux.

De façon générale, la pierre bat les ciseaux (en les émoussant), les ciseaux battent la feuille (en la coupant), la feuille bat la pierre (en l'enveloppant). Ainsi chaque coup bat un autre coup, fait match nul contre le deuxième (son homologue) et est battu par le troisième.

Écrire un programme qui réalise 5 manches du jeu Pierre-feuille-ciseaux entre l'ordinateur et le joueur. Chaque manche va consister en :

— la génération (pseudo) aléatoire d'un nombre entre 0 et 2 compris, à l'aide de la fonction `randint` du module `random`, et qui va représenter le coup de l'ordinateur (0 valant Pierre, 1 Feuille et 2 Ciseaux) ;
— la lecture en entrée (`input`) d'une valeur entière entre 0 et 2 compris qui représente le coup du joueur ;
— l'affichage du résultat sous une des formes :
  — `coup_o` bat `coup_j` : `points`
  — `coup_o` est battu par `coup_j` : `points`
  — `coup_o` annule `coup_j` : `points`

où
— `coup_o` et `coup_j` sont respectivement le coup de l'ordinateur et du joueur : `"Pierre"` s'il a joué 0, `"Feuille"` s'il a joué 1 et `"Ciseaux"` s'il a joué 2.
— `points` donne le résultat des manches jusqu'à présent sachant que le compteur `points` part de zéro, et est incrémenté de un chaque fois que le joueur gagne une manche, et décrémenté de un chaque fois que l'ordinateur gagne une manche (les matchs nuls ne modifiant pas le compteur `points`).

À la fin des cinq manches, votre programme affichera `Perdu`, `Nul` ou `Gagné` suivant que le compteur est négatif, nul ou strictement positif.

Pour plus de clarté dans votre code, nous vous conseillons de définir les trois constantes symboliques :

```
PIERRE = 0
FEUILLE = 1
CISEAUX = 2
```

Par ailleurs, votre code doit importer le module `random` et, **avant de commencer les manches**, pour permettre à UpyLaB de valider les résultats, doit d'abord lire une valeur entière `s` et appeler la fonction `random.seed(s)`. Vous devez donc intégrer le code suivant :

```
import random

PIERRE = 0
FEUILLE = 1
CISEAUX = 2

...

s = int(input())
random.seed(s)
```

Votre code fera donc un appel à `random.seed` suivi de cinq appelq à `random.randint`, un par manche. Aucun autre appel à une fonction de `random` ne pourra être effectué.

Vous pouvez bien sûr utiliser la fonction `bat` de l'exercice 4.9 mais nous vous conseillons vivement de définir aussi d'autres fonctions (par exemple , une fonction qui réalise une manche et imprime la ligne de message) pour structurer votre code.

Exemple 1

Sachant que le code suivant :

```
random.seed(65)
for i in range(5):
    print(random.randint(0,2))
```

donne le résultat :

```
1
1
1
2
0
```

L'exécution du code avec les entrées :

```
65
0
1
2
1
0
```

doit afficher :

```
Feuille bat Pierre : -1
Feuille annule Feuille : -1
Feuille est battu par Ciseaux : 0
Ciseaux bat Feuille : -1
Pierre annule Pierre : -1
Perdu
```

Exemple 2

Sachant que le code suivant :

```
random.seed(1515)
for i in range(5):
    print(random.randint(0,2))
```

donne le résultat :

```
2
1
0
2
2
```

L'exécution du code avec les entrées :

```
1515
0
1
2
1
0
```

doit afficher :

```
Ciseaux est battu par Pierre : 1
Feuille annule Feuille : 1
Pierre bat Ciseaux : 0
Ciseaux bat Feuille : -1
Ciseaux est battu par Pierre : 0
Nul
```

Exemple 3

Sachant que le code suivant :

```
random.seed(2001)
for i in range(5):
    print(random.randint(0,2))
```

donne le résultat :

```
2
0
1
0
0
```

L'exécution du code avec les entrées :

```
2001
0
1
2
1
0
```

doit afficher :

```
Ciseaux est battu par Pierre : 1
Pierre est battu par Feuille : 2
Feuille est battu par Ciseaux : 3
Pierre est battu par Feuille : 4
Pierre annule Pierre : 4
Gagné
```

Consignes

— Dans cet exercice, il vous est demandé d'écrire un programme contenant des fonctions.
— Attention, nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu en respectant majuscules et minuscules et en veillant à n'avoir qu'un espace entre les mots et signes et aucune espace supplémentaire en fin de ligne. En particulier, il ne faut rien écrire à l'intérieur des appels à input (`int(input())` et non `int(input("Entrer un nombre : "))` par exemple), ni ajouter du texte supplémentaire dans ce qui est imprimé (`print(points)` et non `print("résultat :", points)` par exemple).

Quand votre solution sera validée, cliquez sur le bouton « AFFICHER LA RÉPONSE » pour trouver des conseils de bonnes pratiques et éventuellement améliorer votre façon de coder.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une solution à l'exercice : vidéo d'une solution des exercices UpyLaB 4.9 et 4.10 (à partir de 16")

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si rien ne marche : consultez la *FAQ sur UpyLaB 4.10*.

4.6 Bilan du module

4.6.1 Qu'avons-nous vu dans ce module ?

BILAN EN BREF

Note : Voir la vidéo de la section 4.6.1 : Bilan du module 4

BILAN DU MODULE

Nous voici à la fin de ce module. Nous y avons vu principalement comment structurer notre code grâce à la définition et l'utilisation de fonctions. De façon plus précise, nous avons vu :

- que plusieurs fonctions prédéfinies bien utiles sont disponibles ;
- que la librairie standard regorge de modules qui peuvent être importés et sont remplis de fonctions prédéfinies biens utiles également ;
- que le programmeur peut également définir des fonctions Python ;
- que la définition et l'utilisation de fonctions sont essentielles pour écrire un « bon » code, c'est-à-dire un code « bien structuré » facilement « lisible » pour le programmeur et toute personne qui devrait consulter le code ;
- que le mécanisme de passage de paramètres permet d" « isoler » le code d'une fonction du code appelant ;
- qu'il existe toutes sortes de formes de fonctions qui renvoient un résultat ou dont le but est d'effectuer un autre type de traitement ;
- qu'un résultat d'une fonction peut être un tuple de valeurs, ce qui permet en pratique à la fonction de renvoyer plusieurs résultats lors d'un seul `return`.

Nous avons également illustré tous ces concepts avec des exemples et la réalisation d'exercices supervisés ou réalisés de façon autonomes avec UpyLaB.

Dans ce module et les modules précédents, nous avons appris à écrire des codes complets et modulaires. Les deux modules qui vont suivre vont compléter le tableau. Nous allons voir les séquences de données. Ces types de données vont nous permettre de résoudre des problèmes beaucoup plus ambitieux que ce que nous avons fait jusqu'à présent. En route donc vers l'infini et au-delà ! Enfin presque.

CHAPITRE 5

Les séquences de données pour jouer ou travailler

5.1 Manipulations de base des séquences

5.1.1 Présentation du menu de ce module et des chaînes de caractères

MENU DE CE MODULE

Nous voici arrivés au module 5 de notre cours. Ce module est important à la fois au niveau de son contenu et au niveau de son volume.

Nous allons parler de structures de données plus complexes que ce que nous avons vu jusqu'à présent. En clair, nous allons analyser de façon fouillée les trois types de données *chaînes de caractères*, *tuples* et *listes*.

LES CHAÎNES DE CARACTÈRES

Note : Voir la vidéo de la section 5.1.1 : Introduction aux séquences

CONTENU DE LA VIDÉO

La vidéo précédente introduit dans un premier temps le menu de ce module : l'étude des trois séquences Python : les chaînes de caractères, les tuples et les listes.

Nous présentons ensuite les manipulations de base sur les chaînes de caractères dont certaines ont déjà été présentées au module 2.

Code sur la console réalisé dans la vidéo

```
>>> mon_str = "bonjour"
>>> long_t = """*****************************
... * Bon anniversaire           *
... *****************************"""
>>> print(long_t)
*****************************
```

(suite sur la page suivante)

(suite de la page précédente)

```
* Bon anniversaire           *
*****************************
>>> mon_str + " Sebastien"
'bonjour Sebastien'
>>> "ha " * 20
'ha ha ha ha ha ha ha ha ha ha ha ha ha ha ha ha ha ha ha ha '
>>> len(mon_str)
7
>>> mon_str[0]
'b'
>>> mon_str[1]
'o'
>>> mon_str[7]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: string index out of range
>>> mon_str[len(mon_str)-1]
'r'
>>> mon_str[-1]
'r'
>>> mon_str[-3]
'o'
>>>
```

La vidéo en bref

Après avoir présenté le copieux menu de ce module, nous montrons les manipulations de base des chaînes de caractères, parfois plus simplement appelés *textes*. Les constantes chaînes de caractères sont entourées de simples ou doubles apostrophes (appelées également simples ou doubles quotes) ou de trois simples ou doubles quotes ; ces dernières peuvent être multilignes.

Nous rappelons que l'opérateur + sur les chaînes de caractères effectue la concaténation de celles-ci (le fait de coller deux textes ensemble), et que nous pouvons répéter une chaîne avec l'opérateur * suivi ou précédé d'un nombre entier qui donne le facteur de répétition.

Nous montrons que nous pouvons obtenir la valeur d'une composante d'une séquence (ici chaînes de caractères) en indiçant la séquence avec une valeur entière entre 0 et sa longueur moins 1 (par exemple de 0 à 6 pour le texte « bonjour »).

Demander une composante plus grande donne une erreur à l'exécution. Par contre demander la composante d'indice -1, -2, ... revient à demander, respectivement la dernière composante, l'avant dernière, ...

Enfin pour un texte `s`, `len(s)` donne sa longueur, c'est-à-dire le nombre de ses composantes.

5.1.2 Les tuples et les listes

LES TUPLES ET LES LISTES

Note : Voir la vidéo de la section 5.1.2 : Les tuples et les listes

CONTENU DE LA VIDÉO

La vidéo précédente présente les listes et les tuples ainsi que leurs manipulations de base.

Code sur la console réalisé dans la vidéo

```
>>> asteroides = [325, 326, 327, 328, 329, 330]
>>> personnages = ["le serpent", "le mouton", "le petit prince", "l'aviateur", "la rose",
...                "le Monsieur cramoisi", "le roi", "le vaniteux", "le buveur",
...                "le businessman", "l'allumeur de réverbères", "le géographe",
...                "la fleur à trois pétales", "les roses du jardin", "le renard",
...                "l'aiguilleur", "le marchand"]
>>> planetes = ['B 612', 325, 326, 327, 328, 329, 330, 'Terre']
>>> vide = []
>>> mon_str = "Bonjour"
>>> mon_tup = (325, 326, 327, 328, 329, 330)
>>> mon_tup[0]
325
>>> asteroides[3]
328
>>> len(personnages)
17
>>> type(mon_tup)
<class 'tuple'>
>>> type(personnages)
<class 'list'>
>>> type(mon_str)
<class 'str'>

>>>
```

La vidéo en bref

Des listes (`list`) et des tuples (`tuples`), en plus des chaînes de caractères, sont définis et manipulés soit globalement soit en prenant une composante. Les fonctions prédéfinies `len` (qui donne la longueur) et `type` (qui donne le type) sont applicables sur les trois types de séquences.

5.1.3 Tests sur les séquences et instruction for

TESTS ÉLÉMENTAIRES ET INSTRUCTION FOR

Note : Voir la vidéo de la section 5.1.3 : Les séquences : for, in, opérateurs relationnels

CONTENU DE LA VIDÉO

La vidéo précédente revient sur l'instruction `for` qui manipule tous les éléments d'une séquence, présente l'opérateur `in` dans un test (une condition) qui permet de déterminer si une composante d'une séquence vaut une valeur donnée et montre comment l'interpréteur Python compare les séquences entre elles au niveau de l'égalité, de la différence ou de l'ordre.

Code sur la console réalisé dans la vidéo

```
>>> personnages = ["le serpent", "le mouton", "le petit prince", "l'aviateur",
...                "la rose", "le Monsieur cramoisi", "le roi", "le vaniteux",
...                "le buveur","le businessman", "l'allumeur de réverbères",
...                "le géographe", "la fleur à trois pétales", "les roses du jardin",
...                "le renard", "l'aiguilleur", "le marchand"]
>>> somme = 0
>>> for i in range(10):
...     somme += i
...
>>> somme
45
>>> for elem in personnages:
...     print(elem)
...
le serpent
le mouton
le petit prince
l'aviateur
la rose
le Monsieur cramoisi
le roi
le vaniteux
le buveur
le businessman
l'allumeur de réverbères
le géographe
la fleur à trois pétales
les roses du jardin
le renard
l'aiguilleur
le marchand
>>> nom = "Gertrude"
>>> for c in nom:
...     print("Caractère suivant :", c)
...
Caractère suivant : G
Caractère suivant : e
Caractère suivant : r
Caractère suivant : t
Caractère suivant : r
Caractère suivant : u
Caractère suivant : d
Caractère suivant : e
>>> for i in range(len(nom)):
...     print("le caractère d'indice ", i, " :", nom[i])
...
le caractère d'indice  0  : G
le caractère d'indice  1  : e
le caractère d'indice  2  : r
le caractère d'indice  3  : t
le caractère d'indice  4  : r
le caractère d'indice  5  : u
```

(suite sur la page suivante)

(suite de la page précédente)

```
le caractère d'indice  6  : d
le caractère d'indice  7  : e
>>> personne = input("Un personnage : ")
Un personnage : >? Gertrude
>>> if personne in personnages:
...         print(personne, "est un personnage du livre")
...
>>> n1 = "Juliette"
>>> nom
'Gertrude'
>>> nom < n1
True
>>> nom < "Gertruda"
False
>>> nom < "Gert"
False
>>> "Dumoulin" < "Du Pont"
False
>>> prem = [2, 3, 5, 7, 11, 13]
>>> 7 in prem
True
>>> 9 in prem
False
>>> ma_liste = [9, 18, 3, 5, 7, 21]
>>> ma_liste2 = [9, 18, [3, 5, 7], 21]
>>> [3, 5, 7] in ma_liste
False
>>> [3, 5, 7] in ma_liste2
True
>>> "trude" in nom
True
>>> nom
'Gertrude'
```

La vidéo en bref

Des exemples de manipulations de séquences avec des instructions `for` sont donnés. L'opérateur `in` est utilisé dans un for pour itérer sur tous les éléments d'une séquence. L'opérateur `in` peut également être utilisé dans une condition pour tester si un élément est dans une séquence. Son fonctionnement précis ainsi que le fonctionnement des opérateurs relationnels (==, !=, <, <=, >, >=) sont expliqués grâce à divers exemples.

Pour les textes `s` et `t`, `s in t` teste si le texte `s` est une sous-chaîne de la chaîne de caractères `t`. Ce n'est pas le cas pour les tuples ou les listes, et la vidéo explique pourquoi.

5.1.4 Encodage des caractères

CODIFICATION UTILISÉE PAR PYTHON POUR LES TEXTES

Dans les vidéos que nous venons de présenter, nous avons parlé de comparaison entre caractères. Mais, par exemple, quel est le résultat du code ?

```
c1 = '#'
c2 = '€'
if c1 < c2:
    print("Ce code est juste un exemple")
```

Pour le savoir nous devons nous pencher sur la codification des caractères en Python.

En fait, Python encode ces caractères grâce à la codification UTF-8. Regardons ce que cela veut dire.

D'après l'entrée UTF-8 de Wikipedia : « UTF-8 (abréviation de l'anglais Universal Character Set Transformation Format 1 - 8 bits) est un codage de caractères informatiques conçu pour coder l'ensemble des caractères du « répertoire universel de caractères codés » ... Il s'agit du codage le plus utilisé dans les systèmes GNU, Linux et compatibles pour gérer le plus simplement possible des textes et leurs traductions dans tous les systèmes d'écritures et tous les alphabets du monde. »

En clair, à chaque caractère est attribué, par la norme UTF-8, un numéro qui lui est propre et que Python utilise pour le stocker mais aussi pour le comparer aux autres caractères. UTF-8 essaye de donner un numéro à tous les caractères de tous les alphabets connus.

ORD(C)

En Python, la fonction prédéfinie `ord(x)` donne le numéro dans le classement UTF-8 du caractère `x`.

Ainsi `'#' < '€'` est vrai car `ord('#')` vaut 35 et `ord('€')` vaut 8364. Une des seules choses que vous devez retenir est que :

- les caractères qui correspondent aux chiffres `'0'` à `'9'` ont des numéros contigus (`ord('0')` vaut 48, `ord('1')` vaut 49, ...);
- idem pour les lettres minuscules `'a'` .. `'z'` (`ord('a')` vaut 97, `ord('b')` vaut 98, ...), et pour les lettres majuscules `'A'`.. `'Z'` (`ord('A')` vaut 65, ...).

Ainsi ayant un caractère dans la variable `c`, le test Python

```
'a' <= c and c <= 'z'
```

teste bien si `c` est une lettre de l'alphabet en minuscule. On peut faire la même chose avec les caractères majuscules ou les caractères qui correspondent à des chiffres. Attention à ne pas mélanger la valeur `0` et le caractère `'0'` dont la codification UTF-8 (`ord('0')`) vaut 48.

CHR(V)

La fonction prédéfinie `chr(v)` est l'inverse de `ord( )` : `chr(v)` reçoit en paramètre un nombre entier positif `v`, et renvoie la caractère UTF-8 correspondant. Par exemple `chr(65)` renvoie `'A'`.

5.1.5 Illustrons ce que nous avons vu jusqu'à présent sur les séquences

TEMPÊTE OU CYCLONE

Même si nous n'avons pas encore suffisamment vu comment manipuler les séquences pour faire énormément de programmes intéressants, nous allons quand même imaginer deux petits exercices, que nous allons réaliser ensemble, pour illustrer ce que nous avons déjà vu.

D'après l'échelle de Saffir-Simpson, une tempête est définie si les vents enregistrés (vents moyens pendant une minute à une hauteur de 10 mètres) sont de 64 à 118 km/h. Au delà de 118 km/h, selon l'endroit dans le monde où l'on se trouve, on parle d'ouragan, de typhon ou de cyclone.

Les cyclones sont classés suivant leur force, par l'échelle de Saffir-Simpson allant de 1 à 5 :

- Catégorie 1 : Vents de 119 à 153 km/h
- Catégorie 2 : Vents de 154 à 177 km/h
- Catégorie 3 : Vents de 178 à 209 km/h
- Catégorie 4 : Vents de 210 à 250 km/h
- Catégorie 5 : Vents supérieurs à 250 km/h

Un territoire qui subit des vents de catégorie 5 subira des dégâts considérables.

Supposons que l'on ait une liste `mes_valeurs` de valeurs entières positives correspondant aux vents maximums enregistrés aux Bermudes chaque année depuis l'an 2000, la composante 0 désignant l'an 2000, 1, 2001, ...

Par exemple, **avec des chiffres totalement inventés** :

```
mes_valeurs = [75, 200, 230, 260, 100, 80, 50, 45, 180, 100, 200, 63, 64, 65,\
        118, 119, 153, 154, 280, 345]
```

HISTORIQUE_TEMPETES

Écrivons une fonction `historique_tempetes` qui reçoit en paramètre formel la liste `vent_max` de ces valeurs et imprime pour chaque année si les îles ont subi une tempête ou un cyclone et si c'est le cas, de quelle catégorie.

Découpe de la fonction historique_tempetes

La fonction doit traiter chaque élément de la liste séquentiellement. Une instruction `for` est donc adaptée. Pour chaque élément, il faut tester dans quelle catégorie se trouve la valeur. Remarquons que l'on peut découper la valeur en 3 classes : 1) pas de tempête, 2) tempête, 3) cyclone d'une certaine catégorie. Pour ne pas avoir à plusieurs endroits dans le programme du code assez semblable, une **fonction categorie** semble indiquée, qui reçoit une valeur de vent supérieure à 118 km/h et renvoie un nombre entre 1 et 5 suivant la catégorie.

```
def categorie(vent):
    """renvoie la catégorie de cyclone enregistré en fonction du vent"""

    assert vent > 118 # sinon provoque une erreur dans le code

    if vent < 154:
        res = 1
    elif vent < 178:
        res = 2
    elif vent < 210:
        res = 3
    elif vent < 251:
        res = 4
    else: # catégorie 5
        res = 5
    return res
```

La fonction `historique_tempetes` peut par exemple être encodée comme suit :

```
def historique_tempetes(vent_max):
    """affiche pour chaque année la catégorie de vents subis cette année-là
    entrée : vent_max: liste des vents maximums enregistrés en km/h
    chaque année (composante 0 étant pour l'an 2000)"""

    annee = 2000
    for vent in vent_max:
        if vent < 64:
            print("En", annee, ": pas de tempête enregistrée")
        elif vent < 119:
            print("En", annee, ": il y a eu au moins une tempête mais pas de cyclone")
        else:
            print("En", annee, ": le plus gros cyclone enregistré était de catégorie",
            categorie(vent))
        annee += 1
```

qui peut être appelée avec la liste des valeurs :

```
historique_tempetes(mes_valeurs)
```

CYCLONE_ANNEE

Avec la même liste `mes_valeurs` des vitesses de vents maximums, écrivons une autre fonction : `tempete_annee` qui reçoit en paramètre la liste `vent_max` et `annee` entre 2000 et 2018, et qui renvoie la catégorie de cyclone à laquelle ont été confrontées les îles cette année.

Nous supposons comme résultat possible 0 pour aucun cyclone, et une valeur entre 1 à 5 comprises pour donner la catégorie de cyclone.

La fonction peut se définir comme suit :

```
def cyclone_annee(vent_max, annee):
    """renvoie la catégorie de cyclone à laquelle les îles ont été confrontées
    pour l'année annee à partir de 2000)"""

    assert annee >= 2000 # si annee est inférieure à 2000, provoque une erreur

    vent = vent_max[annee - 2000] # valeur de vent pour cette année
    if vent < 119: # pas de cyclone
        res = 0
    else:
        res = categorie(vent) # appel de la fonction categorie pour déterminer la catégorie
    return res
```

qui peut être testée comme suit pour les années 2001 à 2003 et 2009 :

```
for i in (2001, 2002, 2003, 2009):
    print(cyclone_annee(mes_valeurs, i))
```

Ces deux exemples illustrent bien ce que nous avons vu jusqu'ici sur les séquences de données Python. L'activité suivante demande que vous codiez de façon autonome des problèmes qui se résolvent en utilisant des séquences. À vous de jouer !

5.1.6 Mise en pratique des séquences : exercices UpyLaB 5.1 et suivants

EXERCICES UPYLAB

Comme mise en bouche autonome pour ce module, faites l'exercice 5.1 d'UpyLaB. Cet exercice vous demande de mettre en pratique la manipulation simple de tuples, dans ce cas des couples représentant des noms et des prénoms.

Dans ce cours, nous dirons couple pour parler d'un tuple à deux composantes.

Notez que dans le projet, nous utiliserons des couples, c'est-à-dire des tuples à deux valeurs, pour représenter des coordonnées de points dans un espace à deux dimensions.

EXERCICE UPYLAB 5.1 (Vert)

Énoncé

Écrire une fonction `signature` qui reçoit un paramètre `identite`. Ce paramètre est un couple (tuple de deux composantes) dont la première composante représente un nom et la seconde un prénom.

Cette fonction doit retourner la chaîne de caractères formée du prénom suivi du nom, séparés par une espace.

Exemple

L'appel suivant de la fonction :

```
signature(('Massart', 'Thierry'))
```

doit retourner :

```
'Thierry Massart'
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pouvez supposer que l'argument passé à la fonction est valide (couple de deux chaînes de caractères).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(signature(('Hoarau', 'Sébastien')))` par exemple.
— Notez que la fonction *signature* n'accepte qu'un seul paramètre. Ainsi, un entête du type *def signature(nom, prenom)* ne correspond pas à ce qui est attendu, puisque dans ce cas, on définit deux paramètres. Le paramètre atttendu est un tuple, dont on pourra accéder aux composantes dans le corps de la fonction (revoir si besoin la matière correspondante).
— Pensez à utiliser la concaténation de chaînes de caractères.
— Les apostrophes présentes dans le retour de la fonction ne font pas partie de la chaîne de caractères. Elles indiquent que l'objet retourné est de type *str*.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.1*.

5.1.7 Exercice UpyLaB 5.2 (Vert)

Énoncé

On représente un brin d'ADN par une chaîne de caractères qui peut contenir quatre caractères différents : "A"(Adénine), "C" (Cytosine),"G" (Guanine) et "T" (Thymine).

Écrire une fonction `est_adn` qui reçoit une chaîne de caratères en paramètre et qui retourne `True` si cette chaîne de caractères n'est pas vide et peut représenter un brin d'ADN, `False` sinon.

Exemple 1

L'appel suivant de la fonction :

```
est_adn("TAGCAT")
```

doit retourner :

```
True
```

Exemple 2

L'appel suivant de la fonction :

```
est_adn("ISA")
```

doit retourner :

```
False
```

Exemple 3

L'appel suivant de la fonction :

```
est_adn("CTaG")
```

doit retourner :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pouvez supposer que l'argument passé à la fonction sera toujours une chaîne de caractères ; notez que l'appel de la fonction sur une chaîne vide doit retourner `False`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(est_adn(("ADN")))` par exemple.
— Pensez à parcourir la chaîne de caractères passée en argument et tester si chaque caractère se trouve bien dans la chaîne « ACGT ».
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.2*.

5.1.8 Exercice UpyLaB 5.3 (Vert)

Énoncé

Écrire une fonction `duree` qui reçoit deux paramètres `debut` et `fin`. Ces derniers sont des couples (tuples de deux composantes) dont la première composante représente une heure et la seconde les minutes.

Cette fonction doit calculer la durée qui s'est écoulée entre ces deux instants. Le résultat sera donné sous la forme d'un tuple `(heure, minutes)`.

Exemple 1

L'appel suivant de la fonction :

```
duree ((14, 39), (18, 45))
```

doit retourner :

```
(4, 6)
```

Exemple 2

L'appel suivant de la fonction :

```
duree ((6, 0), (5, 15))
```

doit retourner :

```
(23, 15)
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Notez que l'appel `duree ((6, 0), (5, 15))` retourne le couple `(23, 15)` et non `(0, 45)`. Le premier argument correspond à l'instant initial et le second à l'instant final.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(duree ((6, 0), (5, 15)))` par exemple.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.3*.

5.1.9 Exercice UpyLaB 5.4 (Vert)

Énoncé

Écrire une fonction `distance_points()` qui reçoit en paramètres deux tuples de deux composantes représentant les coordonnées de deux points et qui retourne la distance euclidienne séparant ces deux points.

Pour rappel, la distance euclidienne entre les points (x_1, y_1) et (x_2, y_2) se calcule grâce à la formule :

$$dist = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

où $\sqrt{a}$ désigne la racine carrée de a et correspond à $a^{\frac{1}{2}}$.

Exemple 1

L'appel suivant de la fonction :

```
distance_points((1.0, 1.0), (2.0, 1.0))
```

doit retourner :

```
1.0
```

Exemple 2

L'appel suivant de la fonction :

```
distance_points((-1.0, 0.5), (2.0, 1.0))
```

doit retourner (approximativement) :

```
3.0413812651491097
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Il n'est pas demandé que la fonction `distance_points` teste le type des paramètres reçus.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(distance_points((0.0, 0.0), (1.0, 1.0)))` par exemple.
— Notez que la fonction `distance_points` n'accepte que deux paramètres. Ainsi, un entête du type `` def distance_points(x1, y1, x2, y2)`` ne correspond pas à ce qui est attendu, puisque dans ce cas, on définit quatre paramètres. Les deux paramètres attendus sont des tuples, dont on pourra accéder aux composantes dans le corps de la fonction (revoir si besoin la matière correspondante).
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.4*.

5.1.10 Rassembler et séparer

RASSEMBLER

Pour que vous puissiez faire l'exercice UpyLaB suivant, nous devons d'abord introduire deux mécanismes bien pratiques que Python offre en association avec les fonctions : l'opération qui consiste à *rassembler* (*gather* en anglais) et à la sous-section suivante, le mécanisme inverse. Commençons par le *gather* : sans entrer dans tous les détails, dans la définition d'une fonction Python, on peut rajouter une astérisque devant le seul ou le dernier paramètre. Par exemple :

```
def ma_fonction(*valeurs) :
    print("Le paramètre est un tuple valant :", valeurs)
```

Dans ce cas, les valeurs reçues lors de chaque appel sont rassemblées dans un tuple.

Par exemple dans une console Python :

```
>>> def ma_fonction(*valeurs):
 ...       print("Le paramètre est un tuple valant :", valeurs)
 ...
>>> ma_fonction(2,3,5,7)
Le paramètre est un tuple valant : (2, 3, 5, 7)
>>> ma_fonction(2,3)
Le paramètre est un tuple valant : (2, 3)
>>> ma_fonction()
Le paramètre est un tuple valant : ()
>>> ma_fonction(2)
Le paramètre est un tuple valant : (2,)
```

Notez comment un tuple vide ou un tuple d'un élément est dénoté. En particulier :

```
t = (2)
```

mettra l'entier 2 dans `t` (les parenthèses étant considérées comme faisant partie de l'expression arithmétique même si elles ne rajoutent rien).

La virgule dans `(2,)` permet donc de stipuler que l'on parle d'un tuple.

SÉPARER

L'inverse du gather est aussi possible et est appelé en anglais *scatter* (disperser) qui distribue les valeurs d'une séquence comme étant une séquence d'arguments lors de l'appel à une fonction.

Expliquons le *scatter* sur un simple exemple. La fonction prédéfinie `max` renvoie le maximum des valeurs transmises, soit sous forme de liste, soit d'une séquence.

Ainsi analysons dans une console Python, la séquence suivante :

```
>>> max('bonjour')
'u'
>>> max([3, 7, 2, 5])
7
>>> liste1 = [2, 5, 7, 3]
>>> liste2 = [1, 5, 21, -15]
>>> max(liste1, liste2)
[2, 5, 7, 3]
>>> max(*liste1, *liste2)
21
```

- le premier `max` regarde parmi la séquence de lettres celle qui a la valeur maximale (pour la codification UTF-8) ;
- le second, le maximum des valeurs de la liste ;
- le troisième compare deux listes et celle qui commence par 2 est plus grande que celle qui commence par la valeur 1 ;
- la dernière instruction, « met à plat » les valeurs de `liste1` et de `liste2`. Nous obtenons l'équivalent de `max(2, 5, 7, 3, 1, 5, 21, -15)` qui renvoie la valeur maximale soit 21.

À présent vous pouvez effectuer l'exercice UpyLaB 5.5 qui utilise le rassemblement (gather).

5.1.11 Exercice UpyLaB 5.5 (Bleu)

Énoncé

Écrire une fonction `longueur(*points)` qui reçoit en paramètres un nombre arbitraire de points (tuples de deux composantes), et retourne la longueur de la ligne brisée correspondante.

Cette longueur se calcule en additionnant les longueurs des segments formés par deux points consécutifs.

Exemple 1

L'appel suivant de la fonction :

```
longueur((1.0, 1.0), (2.0, 1.0), (3.0, 1.0))
```

doit retourner :

```
2.0
```

Exemple 2

L'appel suivant de la fonction :

```
longueur((0.5, 1.0), (2.0, 1.0), (2.5, -0.5), (-1.5, -1.0))
```

doit retourner (approximativement) :

```
7.1122677042334645
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Pour simplifier, on supposera qu'il y aura toujours au moins deux points passés en paramètre lors des appels à la fonction `longueur`.
— Il est conseillé d'utiliser la fonction `distance_points` définie lors de l'exercice précédent.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(longueur((0.0, 0.0), (1.0, 1.0)))` par exemple.
— Notez que la fonction `longueur` accepte un nombre arbitraire de tuples en arguments, et non une liste ou un tuple de tuples. N'hésitez pas à revoir la matière présentée dans l'onglet de cours précédent.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.5*.

5.2 Slicing et opérations de manipulation des séquences

5.2.1 Tranches, manipulations et séquences imbriquées

REVENONS À NOS SÉQUENCES

Nous voici de retour pour apprendre de la nouvelle matière sur les séquences. Nous avons vu que les séquences permettent de stocker plusieurs valeurs soit de type texte avec les chaînes de caractères, soit de différents types pour les tuples et les listes. Il nous reste à voir beaucoup de matière dans ce gros module sur les séquences, en terme de manipulations plus sophistiquées. En particulier, nous allons voir comment modifier des séquences, par exemple pour ajouter des éléments ou en supprimer.

Continuons donc à découvrir d'autres façons de manipuler les séquences.

Trois capsules vidéo sont proposées ici :
- la première vidéo présente la notion de tranche et comment étendre une liste ;
- la seconde présente la notion de diagramme d'état, qui va nous permettre de bien comprendre comment l'interpréteur Python gère les séquences dans la mémoire du programme ;
- la dernière vidéo présente des séquences imbriquées, et les notions d'emballage et de déballage (pack / unpack).

TRANCHE DE SÉQUENCE

Note : Voir la vidéo de la section 5.2.1 : Découpage en tranches et modifications de listes

CONTENU DE LA VIDÉO

Tranches de séquences (slicing), modifications d'une composante, et extensions de listes (`append` et `extend`) y sont présentés.

Code sur la console réalisé dans la vidéo

```
>>> s = "Bonjour ça va ?"
>>> s[1:3]
'on'
>>> s[:3]
'Bon'
>>> s[1:]
'onjour ça va ?'
>>> prem = [2, 3, 5, 7, 9]
>>> sec = prem[:]
>>> sec
[2, 3, 5, 7, 9]
>>> text = "Bonjour"
>>> text[::2]
'Bnor'
>>> text[::-1]
'ruojnoB'
>>> prem
[2, 3, 5, 7, 9]
>>> prem[4] = 11
>>> prem
[2, 3, 5, 7, 11]
>>> text
'Bonjour'
>>> text[0] = 'b'
Traceback (most recent call last):
```

(suite sur la page suivante)

(suite de la page précédente)

```
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> text = 'b' + text[1:]
>>> text
'bonjour'
>>> mon_tup = (3.14, 3, 66)
>>> mon_tup[1] = 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> personnages = []
>>> nouveau = input("Personnage : ")
Personnage : >? le roi
>>> personnages.append(nouveau)
>>> personnages
['le roi']
>>> nouveau = input("Personnage : ")
Personnage : >? le buveur
>>> personnages.append(nouveau)
>>> personnages
['le roi', 'le buveur']
>>> prem = [2, 3, 5, 7, 11]
>>> prem.extend([13, 17, 19])
>>> prem
[2, 3, 5, 7, 11, 13, 17, 19]
```

La vidéo en bref

Continuons ici la présentation sur les séquences, en montrant les éléments communs aux trois types présentés. Les *tranches* de séquence (slices) sont présentées. Ensuite nous montrons que l'on peut modifier une composante d'une liste contrairement aux chaînes de caractères et tuples. Les méthodes `append` et `extend` sur les listes sont présentées.

Note : Référez-vous à l' aide mémoire pour avoir un résumé ce que fait chaque méthode sur les séquences.

5.2.2 Suite sur les séquences de données

GESTION MÉMOIRE DES SÉQUENCES EN PYTHON

Note : Voir la vidéo de la section 5.2.2 : Opérateurs is et stockage des séquences en mémoire

CONTENU DE LA VIDÉO

Dans cette capsule vidéo, nous parlons d'identité des objets et de l'opérateur `is`. Nous voyons plus précisément les mécanismes d'assignation et de passage de paramètre lors de l'appel à une fonction.

Code sur la console et scripts réalisés dans la vidéo

```
>>> prem = [2,3]
>>> prem = sec = [2,3]
>>> id(prem)
4511685128
>>> id(sec)
4511685128
>>> trois = [2, 3]
>>> id(trois)
4509083592
>>> prem is sec
True
>>> prem is trois
False
>>> prem == trois
True
>>> prem.append(5)
>>> prem
[2, 3, 5]
>>> sec
[2, 3, 5]
>>> trois
[2, 3]
```

```
prem = sec = [1, 3]
trois = [1, 3]
prem[0] = 2
prem.append(5)
prem.extend([7, 11])
t = 'bonjour'
t2 = 'B' + t[1:]
```

```
def fun(s):
    s[1] = 666

ma_liste = [1, 2, 3]
fun(ma_liste)
print(ma_liste)
```

```
def fun(s):
    s = [4, 5, 6]

ma_liste = [1, 2, 3]
fun(ma_liste)
print(ma_liste)
```

La vidéo en bref

L'assignation multiple, l'identificateur d'un objet et le verbe is sont illustrés par le premier code exécuté dans une console PyCharm. Les deux codes suivants, exécutés dans Python Tutor, illustrent le passage d'un paramètre liste lors de l'appel à une fonction, la modification de cette liste dans la fonction et le mécanisme d'assignation qui correspond à donner un ou plusieurs

noms à un objet.

5.2.3 Seconde suite sur les séquences de données

SÉQUENCES IMBRIQUÉES, EMBALLAGE ET DÉBALLAGE (PACK / UNPACK)

Note : Voir la vidéo de la section 5.2.3 : Séquences non simples, tranches, in, emballage et déballage

CONTENU DE LA VIDÉO

Dans cette capsule vidéo, nous affinons des concepts déjà vus et abordons plusieurs nouveaux concepts sur les séquences et sur les listes Python en particulier. Nous montrons des exemples de séquences non simples, c'est-à-dire dont les éléments sont eux-mêmes des séquences. Nous regardons plus en détails ce qui peut être réalisé avec des tranches de liste, comment fonctionne le test `in` et les notions d'emballage et de déballage (pack et unpack en anglais).

Codes sur la console réalisés dans la vidéo

```
>>> personnages = ["le serpent", "le mouton", "le petit prince", "l'aviateur",
 ...                "la rose", "le Monsieur cramoisi", "le roi", "le vaniteux",
 ...                "le buveur","le businessman", "l'allumeur de réverbères",
 ...                "le géographe", "la fleur à trois pétales", "les roses du jardin",
 ...                "le renard", "l'aiguilleur", "le marchand"]
 >>> print(personnages)
 ['le serpent', 'le mouton', 'le petit prince', "l'aviateur", 'la rose', 'le Monsieur cramoisi',
↪'le roi', 'le vaniteux', 'le buveur', 'le businessman', "l'allumeur de réverbères", 'le␣
↪géographe', 'la fleur à trois pétales', 'les roses du jardin', 'le renard', "l'aiguilleur",
↪'le marchand']
 >>> len(personnages)
 17
 >>> personnages[0]
 'le serpent'
 >>> personnages[0][1]
 'e'
 >>> for p in personnages:
 ...     for c in p:
 ...         print(c, end = ' ')
 ...     print()
 ...
 l e   s e r p e n t
 l e   m o u t o n
 l e   p e t i t   p r i n c e
 l ' a v i a t e u r
 l a   r o s e
 l e   M o n s i e u r   c r a m o i s i
 l e   r o i
 l e   v a n i t e u x
 l e   b u v e u r
 l e   b u s i n e s s m a n
 l ' a l l u m e u r   d e   r é v e r b è r e s
 l e   g é o g r a p h e
 l a   f l e u r   à   t r o i s   p é t a l e s
 l e s   r o s e s   d u   j a r d i n
 l e   r e n a r d
 l ' a i g u i l l e u r
```

(suite sur la page suivante)

(suite de la page précédente)

```
l e   m a r c h a n d
>>> s = [1, 3, 3.14, ("bon", "jour"), [5, 7]]
>>> len(s)
5
>>> len(s[3])
2
>>> len(s[3][1])
4
>>> s[4][1] = 666
>>> s
[1, 3, 3.14, ('bon', 'jour'), [5, 666]]
```

```
>>> tex = "bonjour"
>>> prem = [2, 3, 5, 7, 8, 9, 10, 17]
>>> prem[4:7] = [11, 13]
>>> prem
[2, 3, 5, 7, 11, 13, 17]
>>> 7 in prem
True
>>> "o" in tex
True
>>> "jour" in tex
True
>>> [3, 5, 7] in prem
False
>>> sec = [1, [3, 5, 7], 9]
>>> [3, 5, 7] in sec
True
```

```
>>> tp = 2, 3, 5
>>> x, y, z = tp
>>> x
2
>>> y
3
>>> z
5
>>> x, y = 33, 666
>>> x, y = y, x
>>> x
666
>>> y
33
```

La vidéo en bref

Nous illustrons comment manipuler une séquence de données non simple, c'est-à-dire dont les éléments sont eux-mêmes des séquences. Nous utilisons la liste des personnages du petit prince avec des instructions `for` imbriquées ainsi qu'avec un double indiçage, `personnages[0][1]` par exemple. Nous expliquons encore une fois la différence dans l'utilisation du `in` dans les séquences et dans les chaînes de caractères, et montrons ce que donne le *slicing* à gauche d'une assignation (voir `prem[4:7] = [11, 13]`). La notion d'emballage et de déballage est expliquée sur des exemples.

MAIS EN FAIT C'EST QUOI UNE SÉQUENCE PYTHON ?

Avant de mettre en pratique ce que nous avons vu dans les vidéos qui précèdent, revenons brièvement aux notions de base. Comme déjà mentionné au module 2, Python manipule des objets contenant les valeurs et des variables qui sont des noms pour les objets. Ainsi

```
prem = sec = [2, 3]
```

crée un objet de type liste contenant deux sous-objets entiers valant respectivement 2 et 3 ; `prem` et `sec` sont deux noms donnés à cet objet de type liste. En fait le terme Python approprié n'est pas type mais *classe liste*. Les notions de classe et d"objet sont liées au fait que Python est un langage *orienté-objet*. Nous n'en dirons pas beaucoup plus dans ce cours qui n'aborde pas ces concepts avancés de Python.

Ce qui est important à savoir maintenant est qu'une classe, comme la classe liste, est définie avec des **attributs** et aussi des **méthodes**. Nous avons vu dans la première vidéo de cette sous-section les méthodes `append` et `extend`. De façon générale, une méthode réalise un certain traitement sur un objet de cette classe (par exemple `prem.append(666)` qui ajoute une composante à la liste `prem`, ayant la valeur entière 666).

La **syntaxe** pour utiliser une méthode est le *nom de l'objet*, suivi d'un point suivi du *nom de la méthode* et terminé par les arguments entre parenthèses, les parenthèses étant obligatoires même s'il n'y a pas d'arguments.

La section 5.4 est consacrée à l'apprentissage de nombreuses méthodes associées aux séquences ; celles-ci vont nous permettre assez facilement de réaliser une multitude de traitements.

5.2.4 Manipulations simples de séquences

RELEVÉS DES PLUIES COMME ILLUSTRATION DES SÉQUENCES

Illustrons les manipulations de listes sur un exemple complet que nous allons développer ensemble dans cette activité. Ce sera également l'occasion de montrer quelques techniques classiques utilisées par les programmeurs Python.

Supposons que nous voulions récolter les valeurs des précipitations de pluie, pour un mois de l'année, par exemple septembre, pour ensuite pouvoir communiquer ces valeurs à la demande.

Dans cette section, nous partons d'un code `pluie_1.py` sans séquence qui ne répond que très partiellement au problème et construisons le code `pluie_2.py`, qui manipule une séquence de données pour mieux y répondre. Ce sera l'occasion de discuter sur les façons classiques pour initialiser des séquences de données. Avec tout ce bagage en plus, en fin de section nous produisons, avec le code `pluie_3.py`, une version complète et structurée avec des fonctions répondant au problème.

PLUIE_1.PY

Partons d'un programme sans séquences qui calcule les valeurs cumulées de pluies collectées pendant le mois de septembre.

```
""" programme qui somme les pluies pour septembre
   les valeurs quotidiennes sont entrées (input)
   et la somme est affichée en sortie"""
somme_pluie = 0
for i in range(1, 31): # du jour 1 au jour 30 (31 n'est pas inclus!)
   val = int(input("pluie enregistrée le " + str(i) + " septembre :"))
   somme_pluie += val
print("pluie cumulée pour le mois de septembre :", somme_pluie)
print("Au revoir et merci ! ")
```

Code pluie_1.py

Notons :

— l'utilisation de `str()` qui traduit la valeur `i` en texte, associé à l'opérateur de concaténation `+`, il permet de donner un seul paramètre de type chaîne de caractères à la fonction `input()`,

— l'utilisation de l'opérateur += pour cumuler dans la variable `somme_pluie`, au départ initialisée à 0, les valeurs des pluies cumulées,
— le `range(1,31)` dans le `for` pour que l'indice `i` corresponde au jour concerné de septembre.

PLUIE_2.PY

Supposons maintenant que nous désirions enregistrer les valeurs individuelles lues, ainsi que la somme des valeurs cumulées, pour pouvoir faire plus de manipulations de valeurs.

Le code suivant propose une solution en ce sens :

```
""" programme qui enregistre les valeurs quotidiennes
   des pluies pour septembre, entrées (input)
   et affiche en sortie les valeurs de pluie demandées
   (0 pour le cumul des valeurs)"""

# 1) collecte des données
pluie_septembre = [0] * 31 # crée une liste de 31 composantes initialisées à 0
for i in range(1, 31):
   pluie_septembre[i] = int(input("pluie enregistrée le " + str(i) + " septembre :"))
# calcule la somme dans la composante 0
pluie_septembre[0] = sum(pluie_septembre)

# 2) communication des valeurs à la demande
print("Donnez le jour dont vous désirez la valeur des pluies"
    " (entre 1 et 30 pour les pluies du jour, 0 pour avoir tout le mois"
    " et -1 pour terminer)")
jour = int(input("Jour :"))
while jour > -1 and jour < 31:
   print(pluie_septembre[jour])
   jour = int(input("Jour :"))
print("Au revoir et merci ! ")
```

Code pluie_2.py

Notons que :
— le code commence par se créer une variable `pluie_septembre` de type `liste` à 31 composantes (d'indice 0 à 30) contenant des 0 (notons l'utilisation de `[0]* 31` pour cela) ;
— les composantes `i` de `pluie_septembre` sont mises à jour ;
— ensuite la composante 0 reçoit la somme des composantes ;
— enfin, le code renvoie les valeurs demandées jusqu'à ce que l'utilisateur envoie une valeur -1 pour signifier qu'il ne désire plus que le code affiche des valeurs.

LES TECHNIQUES DE LECTURE ET D'INITIALISATION D'UNE SÉQUENCE

Les codes `pluie_1.py` et `pluie_2.py` montrent comment on peut initialiser une liste avec des valeurs lues avec input. Comme la lecture et l'initialisation de séquences de données (chaînes de caractères, tuples ou listes) sont des traitements fréquents en programmation, il est important ici d'ouvrir une parenthèse et de discuter sur quelques techniques classiques pour le faire en Python.

Lecture et initialisation d'une chaîne de caractères

Cette opération est aisée puisque l'instruction `input()` le permet directement avec :

```
s = input()
```

où `s` est la variable qui reçoit la chaîne de caractère lue.

Lecture et initialisation d'un tuple de valeurs

Si l'on suppose que l'utilisateur introduit la séquence de valeurs en mettant une virgule entre chaque valeur le tout sur une seule ligne, l'utilisation de la fonction prédéfinie `eval()` peut être faite.

Regardons ce que fait la fonction `eval` :

`eval(texte)`

évalue le contenu de la chaîne de caractères fournie en argument comme s'il s'agissait d'une expression Python, et renvoie le résultat. Par exemple :

`eval("2 + 2")`

renvoie la valeur entière 4.

Si le texte fourni à `eval` est une chaîne de valeurs séparées par des virgules (par exemple : 2, 3, 5, 7), `eval` renvoie un `tuple`. Dans ce cas, le code suivant assigne à `s` le `tuple` créé par `eval` :

`s = eval(input())`

Si lors de l'exécution de l'instruction ci-dessus, l'utilisateur introduit la ligne :

`"le serpent", "le mouton", "le petit prince", "l'aviateur", "la rose"`

après son exécution, `s` vaudra le tuple :

`("le serpent", "le mouton", "le petit prince", "l'aviateur", "la rose")`

Lecture et initialisation d'une liste de valeurs entières

Les programmes `pluie_1.py` et `pluie_2.py` montrent comment le travail peut être réalisé.

Si la taille de la liste est connue, en gros, le code peut avoir la forme suivante :

```
s = [None] * n
for i in range(n):
   s[i] = int(input())
```

où `n` est une valeur entière donnant le nombre d'éléments à initialiser (`n = 31` par exemple). L'utilisateur du programme tape sur la touche return du clavier après chaque donnée introduite.

Dans ce cas, le code commence par créer une liste avec le bon nombre `n` de composantes, initialisées avec des valeurs quelconques (0, `None`, ...), valeurs qui ne sont pas utilisées et sont changées dans l'instruction `for` qui suit.

Si le nombre d'éléments est inconnu, la méthode `append` peut être utilisée. Dans ce cas, une technique classique est de demander à l'utilisateur d'entrer une donnée à la fois, et à la fin, d'introduire une valeur convenue, appelée généralement valeur sentinelle (par exemple -1 si les données sont toutes positives) pour signifier que la liste des valeurs communiquées est terminée. La forme du code est la suivante :

```
FIN_DES_VALEURS = -1 # valeur sentinelle qui quand elle est lue
                                       # indique la fin des données
...
s = []
i = int(input())
while i != FIN_DES_VALEURS :
   s.append(i)
   i = int(input())
```

La même technique utilisant la fonction prédéfinie `eval()` pour initialiser un tuple peut être employée pour créer une liste. Il suffit que l'utilisateur encode la liste sur une ligne. Par exemple l'exécution de l'instruction :

`s = eval(input())`

avec comme entrée :

`[2, 3, 5, 7]`

initialise `s` avec la liste correspondante.

Création d'une séquence à partir d'une autre

Analysons d'autres cas de figure.

Types différents :

Si les données que le programme désire utiliser pour créer une séquence sont déjà présentes dans une autre séquence `s` d'un autre type, les fonctions prédéfinies `tuple`, `list`, et `str` peuvent être utiles.

Ces fonctions ont comme seul paramètre la séquence préexistante.
- `list(s)` essaye de changer le type du paramètre `s` en liste,
- `tuple(s)` essaye de changer le type du paramètre `s` en tuple,
- `str(s)` crée une chaîne de caractères qui correspond à ce qui serait imprimé par un `print(s)`.

Par exemple, ayant un tuple `s` valant `(2, 3, 5, 7, 9)`

```
t = list(s)
```

assigne à `t` la liste `[2, 3, 5, 7, 9]`

Autre exemple :

```
t = list("bonjour")
```

assigne à `t` une liste de 7 composantes où chacune est un caractère du texte `"bonjour"`

Mêmes types :

Créer une séquence `t` de type tuple ou chaîne de caractères avec une séquence `s` de même type se fait par une simple assignation. Comme les listes sont modifiables, une simple assignation ne suffit en général pas dans ce cas, et il faut faire une copie. La copie de listes sera vue plus en détail en section 5.6.

PLUIE_3.PY

Après cette partie de cours explicative sur l'initialisation de séquences, discutons du programme `pluie_3.py` dont le but est de donner une version complète et structurée du code qui effectue relevé et communication des données de pluie pour une période donnée.

Pour améliorer la structure du code, `pluie_3.py` sera donc composé des deux fonctions :
- `collecte_donnees` qui construit la liste des valeurs
- `affiche_valeur` qui affiche une information demandée ; ajoutons la possibilité de demander les pluies sur une période entre deux jours, par exemple du 3 au 10.

Donnons la fonction `collecte_donnees` qui construit et renvoie la liste :

```
def collecte_donnees(nb_jours):
  """construit et renvoie une liste avec dans la composante i,
     les valeurs entières lues de pluie cumulée du jour i
     et en composante 0, la somme des pluies pour le mois entier
  """

  # réserve la composante 0 pour enregistrer les pluies cumulées du mois
  pluie = [0]

  # reçoit les valeurs
  for i in range(1, nb_jours + 1):  # jours 1 à nb_jours inclu
      pluie.append(int(input("pluie enregistrée le " + str(i) + " du mois :")))

  # calcule la somme dans la composante 0
  pluie[0] = sum(pluie)

  return pluie  # renvoie la liste construite
```

Notons que l'exécution de l'instruction

```
pluie.append(int(input("texte")))
```
- commence par afficher le `texte`,
- ensuite fait l' `input`,

— traduit la chaîne de caractères reçue en `int`,
— ajoute un composant à la liste `pluie`,
— assigne la valeur entière lue à ce nouveau composant.

La liste `pluie` créée est renvoyée au « programme appelant ».

Donnons la fonction `affiche_valeur` qui reçoit la liste avec les valeurs (paramètre formel `pluie`) et la requête ; celle-ci est soit sous la forme d'une valeur entière entre 0 et 31 (30, 29 ou 28 si nous traitons un mois avec moins de jours), 0 étant la valeur cumulée du mois stockée en composante 0, soit d'un couple de valeurs (début, fin) donnant la période dont l'utilisateur veut les pluies cumulées ; `affiche_valeur` affiche l'information demandée avec des instructions `print`.

```
def affiche_valeur(pluie, requete):
  """ Affiche la valeur de pluie demandée
      si requete est un entier entre 1 et 31 : donne la pluie de ce jour
      si requete est un entier valant 0 : donne la pluie cumulée du mois
      si requete est un couple : donne la valeur des pluies cumulées dans l'intervalle
  """

  if type(requete) == int:  # suppose que requête est entre 0 et 31
      # et que pluie[requete] existe
      if requete == 0:  # demande pluie cumulée
          print("Pluie cumulée du mois :", pluie[requete])
      else:
          print("Pluie le", requete, ":", pluie[requete])
  else:  # requete est un couple de valeurs
      print("Pluie cumulée entre le", requete[0], "et le",
            requete[1], ":", sum(pluie[requete[0]: requete[1] + 1]))
```

Notons :

— les tests sur le type et la valeur de la variable `requete` ;
— l'instruction `print("...", sum(pluie[requete[0]: requete[1]+1]))` qui calcule la somme des valeurs de pluie entre `requete[0]` et `requete[1]` incluse et l'affiche avec `print`.

Il nous reste à donner le code principal qui appelle les fonctions `collecte_donnees` et `affiche_valeur`.

```
# main - code principal
pluies_septembre = collecte_donnees(30)  # lecture données
# délivre les informations désirées
print("Donnez le jour 'j' ou la période 'i, j'",
    "dont vous désirez la valeur des pluies cumulées")
print("(0 pour avoir tout le mois et -1 pour terminer)")
jour = eval(input("Jour ou période ? :"))
while jour != -1:
  affiche_valeur(pluies_septembre, jour)
  jour = eval(input("Jour ou période ? :"))
print("Au revoir et merci ! ")
```

Notons l'instruction `jour = eval(input("Jour ou période ? :"))` qui utilise la fonction prédéfinie `eval` comme expliqué plus haut dans cette section. Dans ce cas,

— si l'utilisateur a introduit 7 : jour est un entier,
— si l'utilisateur a introduit 3,10 : jour est un couple (`tuple`) d'entiers.

Code de pluie_3.py

Le code complet du programme `pluie_3.py` est donné ci-dessous :

```
""" programme qui enregistre les valeurs quotidiennes
  des pluies pour septembre, entrées (input)
  et donne ensuite les valeurs à la demande
  et le cumul du mois si on entre 0 comme jour """
# réserve la composante 0 pour enregistrer les pluies cumulées du mois
```

(suite sur la page suivante)

(suite de la page précédente)

```
def collecte_donnees(nb_jours):
  """construit et renvoie une liste avec dans la composante i,
     les valeurs entières lues de pluie cumulée du jour i
     et en composante 0, la somme des pluies pour le mois entier
  """
  # réserve la composante 0 pour enregistrer les pluies cumulées du mois
  pluie = [0]
  # reçoit les valeurs
  for i in range(1, nb_jours + 1):  # jours 1 à nb_jours inclu
      pluie.append(int(input("pluie cumulée le " + str(i) + " du mois :")))
  # calcul la somme dans la composante 0
  pluie[0] = sum(pluie)
  return pluie  # renvoie la liste construite


def affiche_valeur(pluie, requete):
  """affiche la valeur des pluies demandée"""
  if type(requete) == int:  # suppose que requête est entre 0 et 31
      # et que pluie[requete] existe
      if requete == 0:  # demande pluie cumulée
          print("Pluie cumulée du mois :", pluie[requete])
      else:
          print("Pluie le", requete, ":", pluie[requete])
  else:  # requete est un couple de valeurs
      print("Pluie cumulée entre le", requete[0], "et le",
            requete[1], ":", sum(pluie[requete[0]: requete[1] + 1]))


# main - code principal
pluies_septembre = collecte_donnees(30)  # lecture données
# délivre les informations désirées
print("Donnez le jour 'j' ou la période 'i, j'",
    "dont vous désirez la valeur des pluies cumulées")
print("(0 pour avoir tout le mois et -1 pour terminer)")
jour = eval(input("Jour ou période ? :"))
while jour != -1:
  affiche_valeur(pluies_septembre, jour)
  jour = eval(input("Jour ou période ? :"))
print("Au revoir et merci ! ")
```

5.2.5 À vous de jouer avec les exercices UpyLaB 5.6 et suivants !

EXERCICES UPYLAB 5.6 À 5.8

Nous voici au terme de cet exercice qui nous a permis d'illustrer les manipulations vues jusqu'ici de séquences listes, tuples et chaînes de caractères, et également de voir comment initialiser de telles séquences.

Passons à l'apprentissage autonome de la matière vue jusqu'ici sur les séquences.

Réalisez les exercices 5.6 à 5.8 d'UpyLaB. Ces exercices vont vous permettre de manipuler les séquences, leurs composantes, tranches ainsi que les méthodes déjà vues sur ces séquences de données.

EXERCICE UPYLAB 5.6 (Vert)

Énoncé

Le clavier de Gaston dysfonctionne. À chaque fois qu'il tape sur la touche "M", c'est le caractère "%" qui s'affiche.

Écrire une fonction `transcription_clavier(texte)` qui reçoit en paramètre une chaîne caractère produite par le clavier de Gaston, et qui renvoie la chaîne correctement modifiée, à savoir dans laquelle les caractères "%" ont été remplacés par "M".

On supposera que l'argument passé à la fonction sera une chaînes de caractères, ne comportant que des lettres de l'alphabet, des espaces et des éventuels caractères "%".

Exemple

L'appel suivant de la fonction :

```
transcription_clavier('% le %audit')
```

doit retourner :

```
'M le Maudit'
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Il n'est pas demandé que la fonction teste le type de l'argument passé, ni même que cet argument corresponde à un texte valide.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(transcription_clavier('Bonjour %onsieur %'))` par exemple.
— N'oubliez pas qu'une chaîne de caractères est une séquence non modifiable.

5.2.6 Exercice UpyLaB 5.7 (Vert)

Énoncé

Écrire une fonction `plus_grand_bord(w)` qui reçoit un mot `w` et retourne le plus grand bord de ce mot.

On dit qu'un mot `u` est un bord du mot `w` si `u` est à la fois un préfixe strict et un suffixe strict de `w`, c'est-à-dire qu'on retrouve le mot `u` au début et à la fin du mot `w`, sans que `u` soit égal à `w` lui-même.

Exemples : `'a'` et `'abda'` sont des bords de `'abdabda'`. En effet, `'abdabda'` commence et se termine par `'a'`, ainsi que par `'abda'`. Le plus grand bord de `'abdabda'` est `'abda'`.

Si `w` n'a pas de bord, la fonction retourne la chaîne de caractères vide.

Exemple 1

L'appel suivant de la fonction :

```
plus_grand_bord('abdabda')
```

doit retourner :

```
'abda'
```

Remarque : Notez que les apostrophes indiquent ici que l'objet retourné est de type str. Ces apostrophes ne font pas partie de la chaîne de caractères elle-même.

Exemple 2

L'appel suivant de la fonction :

```
plus_grand_bord('abcabd')
```

doit retourner :

```
''
```

Exemple 3

L'appel suivant de la fonction :

```
plus_grand_bord('abcba')
```

doit retourner :

```
'a'
```

Exemple 4

L'appel suivant de la fonction :

```
plus_grand_bord('aaaaa')
```

doit retourner :

```
'aaaa'
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(plus_grand_bord('abdabda'))` par exemple.
— Notez que le bord d'un mot se retrouve à l'identique au début et à la fin de ce mot, sans symétrie. Si le préfixe est `'abda'`, on doit retrouver le suffixe `'abda'` et non `'adba'`.
— Pour un mot de longueur 5, par exemple, le plus grand bord aura une taille maximale de 4 caractères. On pourra donc tester si la sous-chaîne composée des 4 premiers caractères se retrouve pas à la fin du mot. Si ce n'est pas le cas, il faut alors regarder la sous-chaîne formée des 3 premiers caractères, puis la sous-chaîne formée des 2 premiers caractères, et ainsi de suite.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.7*.

5.2.7 Exercice UpyLaB 5.8 (Bleu)

Énoncé

Écrire une fonction `prime_numbers` qui reçoit comme paramètre un nombre entier `nb` et qui renvoie la liste des `nb` premiers nombres premiers.

Si le paramètre n'est pas du type attendu, ou ne correspond pas à un nombre entier positif ou nul, la fonction renvoie `None`.

Exemple 1

L'appel suivant de la fonction :

```
prime_numbers(4)
```

doit retourner :

```
[2, 3, 5, 7]
```

Exemple 2

L'appel suivant de la fonction :

```
prime_numbers(-2)
```

doit retourner :

```
None
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes (ici par exemple, une fonction qui détermine si le nombre passé en argument est premier ou non), et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— On rappelle qu'un nombre premier est un entier naturel qui possède exactement deux diviseurs distincts et positifs, 1 et lui-même.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(prime_numbers(5))` par exemple.
— N'oubliez pas, lors de ces tests dans votre IDE, d'appeler votre fonction avec un argument invalide ; cela ne doit occasionner aucune erreur, la fonction retourne `None` dans ces cas-là.
— On pourra utiliser la fonction `premier` de l'exercice UpyLaB 4.3.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.8*.

5.3 Testons vos connaissances sur les séquences

5.3.1 Quiz sur les séquences

MINI QUIZ SUR LES SÉQUENCES

Avant de continuer l'étude des séquences Python, ce quiz va vous aider à vérifier que vous avez bien assimilé ce que nous avons vu jusqu'ici sur cette matière.

Note : Voir à la section 5.3.1 du cours en ligne

DIAGRAMME D'ÉTAT

Il est intéressant de voir, en exécutant avec Python Tutor le code de la question 6 ci-dessous, les diagrammes d'état qui montrent que chaque composante de la liste globale construite pointe vers la même sous-liste `[2, 3]` :

Note : animation : voir cours en ligne

5.4 Méthodes de manipulation des séquences

5.4.1 Présentation d'autres méthodes de manipulation des séquences

AIDE-MÉMOIRE

Nous avons vu comment définir et manipuler des séquences, leurs composantes et tranches. Pour les listes, nous avons aussi vu les méthodes `append` et `extend` pour les étendre avec de nouveaux éléments. Python fournit de nombreuses méthodes pour manipuler les trois types de séquences que nous étudions dans ce module.

Dans les capsules vidéo qui suivent, nous allons terminer l'étude des séquences de données chaînes de caractères, tuples et listes Python 3 et leurs manipulations, en présentant un bon nombre de ces méthodes. Elles vont nous permettre de résoudre de nombreux problèmes.

Pour nous faciliter la tâche dans la conception de nos codes, nous avons confectionné un petit aide-mémoire téléchargeable ici résumant l'effet de chacune des fonctions et méthodes présentées ou que nous allons présenter dans la suite de ce cours.

Nous pourrons ainsi décider quelles méthodes peuvent nous aider, et éventuellement si nous devrons écrire d'autres fonctions de manipulation pour résoudre les problèmes soumis.

Note : Quand la syntaxe d'une instruction ou expression Python est donnée la méta-notation `"[...]"` signifie que cette partie est optionnelle. Ainsi :

`s.count(sub [,start [,end]])`

signifie que `s.count` peut avoir un, deux ou trois arguments.

Ainsi :

- `"Bonjour UpyLaB!".count('o')` renvoie 2 (le caractère `'o'` est 2 fois dans `"Bonjour UpyLaB!"`)
- `"Bonjour UpyLaB!".count('o', 3)` renvoie 1 (le caractère `'o'` est 1 fois dans le texte `"Bonjour UpyLaB!"[3:]`)
- `"Bonjour UpyLaB!".count('o', 3, 4)` renvoie 0 (le caractère `'o'` n'est pas dans le texte `"Bonjour UpyLaB!"[3:4]` (rappelez-vous que dans ce cas, l'indice 4 est non inclus).

Comme pour le manuel de bonnes pratiques, pourquoi ne pas imprimer cet *aide-mémoire* pour toujours l'avoir sous la main quand vous écrirez du code ?

N'hésitez pas à le consulter et à tester par vous-même l'effet de chacune d'elles.

RECHERCHE ET COMPTAGE - ZIP ET ENUMERATE

Note : Voir la vidéo de la section 5.4.1

CONTENU DE LA VIDÉO

Cette vidéo présente des méthodes de recherche et de comptage sur les séquences ainsi que les fonctions `zip` et `enumerate`.

Codes sur la console réalisés dans la vidéo

```
>>> min("bonjour")
'b'
>>> sum([2, 5, 6])
13
>>> sum("bonjour")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> prem = [2, 3, 5, 7, 11]
>>> 7 in prem
True
>>> prem.index(7)
3
>>> prem.index(9)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: 9 is not in list
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> x = 7
>>> if x in prem:
...     print(prem.index(x))
...
3
```

```
>>> amis = ["Sébastien", "Pierre", "Ariane", "Thierry"]
>>> couleurs = ["bleu", "rouge", "vert", "jaune"]
>>> for i, a in enumerate(amis):
...     print(i+1, a)
...
1 Sébastien
2 Pierre
3 Ariane
4 Thierry
>>> for a, c in zip(amis, couleurs):
...     print(a, "aime le ", c)
...
Sébastien aime le  bleu
Pierre aime le  rouge
Ariane aime le  vert
Thierry aime le  jaune
```

La vidéo en bref

Les méthodes `min`, `sum`, `index`, `enumerate` et `zip` sont présentées. Référez-vous à l' *aide-mémoire* pour un résumé de ce qu'elles font.

5.4.2 Manipulation de chaînes de caractères

MANIPULATION DE CHAÎNES DE CARACTÈRES

Note : Voir la vidéo de la section 5.4.2

CONTENU DE LA VIDÉO

Cette vidéo présente des méthodes de manipulation spécifiques aux chaînes de caractères.

Code sur la console réalisé dans la vidéo

```
>>> s = 'Bonjour'
>>> s.lower()
'bonjour'
>>> s
'Bonjour'
>>> s = s.lower()
>>> s.upper()
'BONJOUR'
>>> s.islower()
True
>>> s.isdigit()
```

(suite sur la page suivante)

(suite de la page précédente)

```
False
>>> s.isalnum()
True
>>> s.isalpha()
True
>>> help('str')
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 ...

>>> s.find('o')
1
>>> s
'bonjour'
>>> s.find('o', 2)
4
>>> s = "madame Claude rencontre madame Dominique"
>>> s.replace('madame', 'monsieur')
'monsieur Claude rencontre monsieur Dominique'
>>> s
'madame Claude rencontre madame Dominique'
>>> s.capitalize()
'Madame claude rencontre madame dominique'
>>> s = 'bOnjouR'
>>> s.capitalize()
'Bonjour'
>>> s.strip()
'bOnjouR'
```

La vidéo en bref

Des méthodes de manipulation de chaînes de caractères sont présentées : `lower()`, `upper()`, `islower()`, `isdigit()`, `isalnum()`, `isalpha()`, `find()`, `replace()`, `capitalize()`, `strip()`.

Référez-vous à l' *aide-mémoire* pour un résumé de ce qu'elles font.

5.4.3 Manipulation de listes

MANIPULATION DE LISTES

Note : Voir la vidéo de la section 5.4.3

MODIFICATION EN PLACE

Dans les sections précédentes, nous avons parlé de fonctions, puis nous avons introduit le terme de **méthode** pour parler de fonction propre à un objet avant de présenter plusieurs de ces méthodes pour l'objet chaîne de caractères. Les chaînes de caractères étant immuables, les méthodes les manipulant ne peuvent les modifier. Par exemple si la variable `s` référence la chaîne `'python'` alors `s.upper()` crée un nouvel objet `'PYTHON'`.

Avec les listes, nous avons vu des méthodes qui modifient l'objet qui réalise l'appel. On parle de **modification en place**. C'est le cas par exemple de la méthode `sort`. Si `ma_liste` vaut `[3, 2, 1]` alors après l'appel `ma_liste.sort()`, `ma_liste` vaudra `[1, 2, 3]`. Parfois on peut ne pas vouloir modifier son objet initial lors du tri. On pourra alors utiliser la fonction `sorted` qui crée un nouvel objet. Ainsi, toujours avec `ma_liste`, `sorted(ma_liste)` crée une nouvelle liste `[1, 2, 3]` mais ne modifie pas `ma_liste`.

Les méthodes `insert`, `append` sont d'autres méthodes qui modifient en place la liste appelante. N'hésitez pas, lorsque vous croisez de nouvelles méthodes sur des objets mutables comme les listes à vous poser la question : est-ce une modification en place ou non ? Si oui et si vous ne voulez pas modifier votre objet alors il faut rechercher une fonction équivalente qui crée un nouvel objet… ou l'écrire, si elle n'existe pas.

CONTENU DE LA VIDÉO

Cette vidéo présente des méthodes de manipulation spécifiques aux listes

Code sur la console réalisé dans la vidéo

```
>>> prem = [7, 3, 11, 2, 9, 12, 14, 16, 15]
>>> prem.insert(2,5)
>>> prem
[7, 3, 5, 11, 2, 9, 12, 14, 16, 15]
>>> prem.pop(5)
9
>>> prem
[7, 3, 5, 11, 2, 12, 14, 16, 15]
>>> prem.pop()
15
>>> prem.remove(14)
>>> prem
[7, 3, 5, 11, 2, 12, 16]
>>> prem.reverse()
>>> prem
[16, 12, 2, 11, 5, 3, 7]
>>> prem.sort()
>>> prem
[2, 3, 5, 7, 11, 12, 16]
>>> l = prem.copy()
>>> l2 = prem[:]
>>> del prem[7]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list assignment index out of range
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> prem
[2, 3, 5, 7, 11, 12, 16]
>>> del prem[5]
>>> prem
[2, 3, 5, 7, 11, 16]
>>> l = [3, 1, 2]
>>> sorted(l)
[1, 2, 3]
>>> l
[3, 1, 2]
>>> l.sort()
>>> l
[1, 2, 3]
>>> prem
[2, 3, 5, 7, 11, 16]
>>> prem[1:3]
[3, 5]
>>> prem[1:3] = [666, 777]
>>> prem
[2, 666, 777, 7, 11, 16]
>>> prem[1:2] = [666, 777]
>>> prem
[2, 666, 777, 777, 7, 11, 16]
>>> prem[3:3] = [444, 555]
>>> prem
[2, 666, 777, 444, 555, 777, 7, 11, 16]
>>> prem[0:0] = [111, 222]
>>> prem
[111, 222, 2, 666, 777, 444, 555, 777, 7, 11, 16]
>>> prem[len(prem): len(prem)] = [888, 999]
>>> prem
[111, 222, 2, 666, 777, 444, 555, 777, 7, 11, 16, 888, 999]
>>> prem[len(prem)] = 0
Traceback (most recent call last):
  File "<input>", line 1, in <module>
IndexError: list assignment index out of range
>>> prem[len(prem): len(prem)] = [0]
>>> prem
[111, 222, 2, 666, 777, 444, 555, 777, 7, 11, 16, 888, 999, 0]
```

La vidéo en bref

Des méthodes de manipulation de listes sont présentées : `insert()`, `pop()`, `remove()`, `reverse()`, `sort()`, `copy()` ; ainsi que l'instruction `del` et la fonction `sorted`.

Référez-vous à l' *aide-mémoire* pour un résumé de ce qu'elles font.

L'utilisation de tranches (slicing) pour modifier les listes est également illustrée.

5.4.4 Exemples simples qui utilisent des méthodes sur des séquences

TROIS EXEMPLES

Pour digérer la matière vue dans cette section, nous développons ensemble trois exemples qui utilisent méthodes et fonctions prédéfinies sur les chaînes de caractères et listes. À nouveau pour chaque problème, nous vous invitons à écrire une solution. Ensuite, comparez votre solution avec celle proposée pour valider ou trouver d'autres techniques.

NETTOYAGE D'UNE CHAÎNE DE CARACTÈRES

La fonction `keep_alnum` retourne la chaîne passée en argument, en lui ayant retiré tous les caractères qui ne sont pas des lettres ou des chiffres. Elle utilise un `for` pour itérer sur chaque caractère et un `if` qui teste s'il doit être retenu.

```
def keep_alnum(s):
  """Nettoie s en lui retirant les caractères non alphanumériques."""

  res = ''
  for letter in s:
     if  letter.isalnum():
        res = res + letter
  return res
```

L'appel :

```
keep_alnum('he23.?56th')
```

donne comme résultat :

```
'he2356th'
```

Nous verrons à la section 5.6 qu'une version plus compacte et plus simple de `keep_alnum` peut être définie avec la notion de *compréhension de liste*.

LISTE DES LETTRES COMMUNES À 2 MOTS

Tout est dans le titre.

Voici une proposition de code qui résout le problème :

```
def in_both(word1, word2):
 """Renvoie la liste des lettres communes aux 2 mots."""

 res=[]
 for letter in word1:
   if letter in word2 and letter not in res:
     res.append(letter)
 return res
```

Avec cette définition, l'instruction :

```
print(in_both('pommeees', 'oranges'))
```

imprime :

```
['o', 'e', 's']
```

Ici, il faudra attendre le module suivant pour voir une version plus compacte.

LECTURE D'UNE LISTE DE VALEURS (ENTIÈRES)

Une autre version de lecture de liste de valeurs. Ici, la fonction demande à l'utilisateur d'encoder une liste de valeurs entières sur une ou plusieurs lignes, terminée par une ligne vide (touche clavier return sans rien avant), et renvoie la liste des nombres entiers lus.

La méthode `split` est utilisée pour décortiquer les lignes lues.

```
def lecture(invite):
  """Lit et renvoie une liste d'entiers.
   Les données peuvent être sur plusieurs lignes.
  """

  res = []
  x = input(invite)
  while x != '':
     decoupe = x.split()
       # liste des parties de x séparées par une/des espace(s) ou tab
     for elem in decoupe:
        res.append(int(elem))
     x = input()
  return res

# utilisation de la fonction lecture
print(lecture("liste d'entiers terminée par une ligne vide : "))
```

Essayez ce code par exemple avec les entrées sur 5 lignes dont la dernière vide :

```
2 3 5
7 11
13
17 19
```

Passons maintenant à la mise en pratique Upylab avec ou sans méthode !

5.4.5 Mise en pratique : exercices Upylab 5.9 et suivants avec ou sans méthode !

EXERCICES UPYLAB 5.9 (Vert)

Énoncé

Une anagramme d'un mot `v` est un mot `w` qui comprend les mêmes lettres que le mot initial `v`, en même quantité, mais non nécessairement dans le même ordre (par exemple, « marion » et « romina » sont des anagrammes). Notez que anagramme est un mot féminin.

Écrire une fonction `anagrammes(v, w)` qui renvoie la valeur booléenne `True` si les mots `v` et `w` sont des anagrammes.

La fonction retourne la valeur booléenne `False` dans le cas contraire.

Exemple 1

L'appel suivant de la fonction :

```
anagrammes('marion', 'romina')
```

doit retourner :

```
True
```

Exemple 2

L'appel suivant de la fonction :

```
anagrammes('bonjour', 'jour')
```

doit retourner :

```
False
```

Exemple 3

L'appel suivant de la fonction :

```
anagrammes('pate', 'patte')
```

doit retourner :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Votre code ne doit pas tester si les mots existent bien dans un quelconque dictionnaire.
— Pour simplifier, on supposera que tout mot est une anagramme de lui-même (par exemple, `anagrammes('jour', 'jour')` renvoie `True`), et on ne considèrera que des mots écrits en minuscule. Par ailleurs, il n'est pas demandé que votre fonction teste si les mots sont recensés dans un dictionnaire quelconque.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(anagrammes('marion', 'romina'))` par exemple.
— Pensez bien à tester avec les différents cas possibles, référez-vous en particulier aux exemples proposés.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.9*.

5.4.6 Exercice UpyLaB 5.10 (Vert)

Énoncé

Écrire une fonction `dupliques` qui reçoit une séquence en paramètres.

La fonction doit renvoyer la valeur booléenne `True``si elle contient des éléments dupliqués, et la valeur booléenne ``False` sinon.

Exemple 1

L'appel suivant de la fonction :

```
dupliques([1, 2, 3, 4])
```

doit retourner :

```
False
```

Exemple 2

L'appel suivant de la fonction :

```
dupliques(['a', 'b', 'c', 'a'])
```

doit retourner :

```
True
```

Exemple 3

L'appel suivant de la fonction :

```
dupliques('abcda')
```

doit retourner :

```
True
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Il n'est pas demandé que la fonction teste le type de l'argument ; vous pouvez supposer qu'il s'agira bien d'une séquence.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(dupliques(['a','b', 'c', 'd']))` par exemple.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.10*.

5.4.7 Exercice UpyLaB 5.11 (Bleu)

Énoncé

Écrire une fonction `intersection(v, w)` qui calcule l'intersection entre deux chaînes de caractères `v` et `w`.

On définit l'intersection de deux mots comme étant la plus grande partie commune à ces deux mots. Par exemple, l'intersection de « programme » et « grammaire » est « gramm ».

Si les deux chaînes n'ont aucun caractère en commun, la fonction retourne la chaîne vide, `''`.

Si plusieurs solutions sont possibles, la fonction retournera la sous-chaîne d'indice minimal dans `v`. Par exemple, `intersection('bbaacc', 'aabb')` renvoie `'bb'`.

Exemple 1

L'appel suivant de la fonction :

```
intersection('programme', 'grammaire')
```

doit retourner :

```
'gramm'
```

Exemple 2

L'appel suivant de la fonction :

```
intersection('salut', 'merci')
```

doit retourner :

```
''
```

Exemple 3

L'appel suivant de la fonction :

```
intersection('merci', 'adieu')
```

doit retourner :

```
'e'
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(intersection('thierry', 'sébastien'))` par exemple.
— On peut penser à parcourir toutes les sous-chaînes de `v` et vérifier leur présence dans `w`.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.11*.

5.4.8 Exercice UpyLaB 5.12 (Bleu)

Énoncé

Écrire une fonction `my_insert` qui reçoit comme premier paramètre une liste d'entiers relatifs triée par ordre croissant et comme deuxième paramètre un entier relatif `n`, et qui renvoie une liste correspondant à la liste reçue, mais dans laquelle le nombre `n` a été inséré à la bonne place.

La liste passée en paramètre ne doit pas être modifiée par la fonction.

Vous pouvez supposer que le premier paramètre sera bien une liste triée d'entiers, mais si le deuxième paramètre n'est pas du bon type, la fonction retourne `None`.

Exemple 1

L'appel suivant de la fonction :

```
my_insert([1, 3, 5], 4)
```

doit retourner :

```
[1, 3, 4, 5]
```

Exemple 2

L'appel suivant de la fonction :

```
my_insert([2, 3, 5], 1)
```

doit retourner :

```
[1, 2, 3, 5]
```

Exemple 3

L'appel suivant de la fonction :

```
my_insert([2, 3, 5], 0.5)
```

doit retourner :

```
None
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(my_insert([2, 6, 12, 15], 9))` par exemple.
— N'oubliez pas, lors de ces tests dans votre IDE, d'appeler votre fonction avec un argument invalide ; cela ne doit occasionner aucune erreur et la fonction retourne `None` dans ces cas-là.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.12*.

5.4.9 Exercice UpyLaB 5.13 (Rouge)

Énoncé

L'exercice est le même que le précédent, mais ici, si les paramètres ont le type attendu, la fonction modifie la liste en place et ne retourne rien. Si les paramètres ne sont pas valides, une erreur se produit à l'exécution.

Exemple 1

L'exécution du code suivant :

```
l = [1, 3, 5]
my_insert(l, 4)
print(l)
```

doit afficher :

```
[1, 3, 4, 5]
```

Exemple 2

L'exécution du code suivant :

```
l = [1, 3, 5]
my_insert(l, 'a')
print(l)
```

doit provoquer une exception Python, par exemple :

```
AssertionError
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Le code que vous soumettez à UpyLaB ne doit pas non plus contenir la définition de la variable `l`.
- Pour tester le type des paramètres, vous pouvez utiliser les verbes `assert`, `type` ou `isinstance`.
  - l'instruction `assert condition` laisse continuer l'exécution du code si la condition évaluée est vraie, mais provoque une erreur (appelée exception en Python) si la condition est fausse.
  - `type(v)` donne le type de `v`.
  - `isinstance(v, typ)` teste si `v` est de type `typ`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son effet, sur plusieurs valeurs. N'hésitez pas à reproduire les exemples donnés ci-dessus.
- N'oubliez pas, lors de ces tests dans votre IDE, d'appeler votre fonction avec un argument invalide ; cela doit cette fois occasionner une erreur.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 5.13*.

5.4.10 Exercice UpyLaB 5.14 (Vert)

Énoncé

(D'après une idée de Jacky Trinh - 11/02/2018)

Nous pouvons définir la distance entre deux mots de même longueur (c'est-à-dire ayant le même nombre de lettres) `mot_1` et `mot_2` comme le nombre minimum de fois où il faut modifier une lettre de `mot_1` pour obtenir `mot_2`, sans changer leur ordre (distance de Hamming).

Par exemple, les mots « lire » et « bise » sont à une distance de 2, puisqu'il faut changer le "l" et le "r" du mot « lire » pour obtenir « bise ».

Écrire une fonction `distance_mots(mot_1, mot_2` qui retourne la distance entre deux mots.

Vous pouvez supposer que les deux mots sont de même longueur, et sont écrits sans accents.

Exemple 1

L'appel suivant de la fonction :

```
distance_mots("lire", "bise")
```

doit retourner :

```
2
```

Exemple 2

L'appel suivant de la fonction :

```
distance_mots("Python", "Python")
```

doit retourner :

```
0
```

Exemple 3

L'appel suivant de la fonction :

```
distance_mots("merci", "adieu")
```

doit retourner :

```
5
```

Exemple 4

L'appel suivant de la fonction :

```
distance_mots("niche", "chien")
```

doit retourner :

```
5
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(distance_mots("lire", "bise"))` par exemple.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.14*.

5.4.11 Exercice UpyLaB 5.15 (Rouge)

Énoncé

(D'après une idée de Jacky Trinh - 11/02/2018)

Joao vient d'arriver dans notre pays depuis le Portugal. Il a encore du mal avec la langue française. Malgré ses efforts considérables, il fait une faute d'orthographe quasi à chaque mot. Son souci est qu'il n'arrive pas toujours à écrire un mot correctement sans se tromper à une lettre près. Ainsi pour écrire « bonjour », il peut écrire « binjour ». Pour remédier à ce problème, Joao utilise un correcteur orthographique. Malheureusement, Joao a un examen aujourd'hui et il a oublié son petit correcteur.

Afin de l'aider, nous vous demandons **d'écrire une fonction** `correcteur(mot, liste_mots)` où `mot` est le mot que Joao écrit et `liste_mots` est une liste qui contient les mots (ayant la bonne orthographe) que Joao est susceptible d'utiliser.

Cette fonction doit retourner le mot dont l'orthographe a été corrigée.

Exemple 1

L'appel suivant de la fonction :

```
correcteur("bonvour", ["chien", "chat", "train", "voiture", "bonjour", "merci"])
```

doit retourner :

```
"bonjour"
```

Exemple 2

L'appel suivant de la fonction :

```
correcteur("chat", ["chien", "chat", "train", "voiture", "bonjour", "merci"])
```

doit retourner :

```
"chat"
```

Consignes

— Pour cet exercice, vous pourrez utiliser la fonction `distance_mots(mot_1, mot_2)` que vous avez précédemment codée et qui donne la distance entre deux mots de même longueur. N'oubliez pas alors de mettre aussi le code de cette fonction dans votre solution.
— Le correcteur orthographique demandé est une version simple ; les mots en paramètre auront au maximum une seule lettre qui diffère par rapport à la bonne orthographe.
— Nous ne prenons pas en compte les mots avec accents, ni les mots composés de tiret, d'apostrophes, d'espace,..
— `liste_mots` ne contient pas de mots qui se ressemblent ; si Joao écrit le mot « liee », il se peut en effet que cela représente le mot « lire » ou le mot « lier ». Afin d'éviter cette confusion, deux mots de même longueur de la liste sont au moins à une distance de 3. Il n'y aura ainsi qu'un seul mot dans `liste_mots` répondant au problème.
— Vous pouvez supposer que Joao soit arrive à écrire des mots sans fautes, soit fait au plus une erreur.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Notez que les mots de `liste_mots` ne sont pas tous de la même longueur.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.15*.

5.5 Testons nos connaissances

5.5.1 Quiz sur les séquences

ENCORE UN MINI QUIZ SUR LES SÉQUENCES

Note : Voir le quiz de la section 5.5.1

5.6 Compréhensions et copies de listes

5.6.1 Concepts de compréhension et copie profonde de liste

COMPRÉHENSION ET COPIE PROFONDE

Note : Voir la vidéo de la section 5.6.1 : Compréhension et copie profonde (deepcopy) sur les listes

CONTENU DE LA VIDÉO

La vidéo précédente introduit la construction de listes grâce à la technique Python de compréhension de listes.

Elle introduit également grâce à des exemples la nécessité de faire des **copies profondes** (deepcopy) de listes non simples quand les **copies superficielles** (shallow copy) ne sont pas suffisantes.

Code sur la console ou dans Python Tutor réalisé dans la vidéo

```
>>> squares = [x**2 for x in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> m = [(x, y) for x in range(1, 4) for y in range(1, 4) if x != y]
>>> m
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> m2 = []
>>> for x in range(1,4):
...     for y in range(1, 4):
...         if x != y:
...             m2.append((x,y))
...
>>> m2
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

Note : Voir cours en ligne pour l'animation

```
s= [2, 3, 5, 7]
t = s[:]
s[0] = 666
print(s)
print(t)

s = [2, 3, [5, 7]]
t = s[:]
s[0] = 999
s[2][0] = 666
print(s)
print(t)
```

```
from copy import deepcopy
s = [2, 3, [5, 7]]
t = deepcopy(s)
s[2][0] = 666
print(s)
print(t)
```

La vidéo en bref

Deux exemples de compréhension de listes sont donnés, la première simple, la suivante avec deux `for` imbriqués et une condition `if`. La problématique des copies superficielles de listes est ensuite illustrée à travers une exécution dans Python Tutor. L'utilité de la copie profonde d'une liste non simple est ensuite expliquée à nouveau en utilisant Python Tutor pour bien voir les allocations mémoire des différentes parties des séquences de données.

5.6.2 Exemples avec compréhensions de listes

LECTURE D'UNE LISTE D'ENTIERS

La technique de compréhension de liste nous permet de construire facilement une liste de valeurs lues (comme dans l'exemple ci-dessous où l'on suppose que les valeurs sont entières).

Dans le code suivant, l'utilisateur encode sur une ligne une liste de valeurs entières séparées d'au moins une espace (par exemple `1 2 3 21 36 <return>`); ces valeurs permettent d'initialiser la liste `ma_liste` :

```
texte = input('liste de valeurs')
ma_liste = texte.split()  # découpe le texte en parties
                          # séparées d'au moins une espace
ma_liste = [int(i) for i in ma_liste] # crée la liste
                          # en traduisant chaque partie en entier
```

Un code équivalent peut être écrit en une ligne : il effectue séquentiellement l' `input`, le `split` et enfin la construction de la liste avec traduction des données en `int` :

```
liste = [int(i) for i in input("liste des valeurs").split()]
```

KEEP_ALNUM VERSION 2

Reprenons la fonction `keep_alnum` que nous avons présentée dans la sous-section 5.4.4, et dont le but est de renvoyer un texte nettoyé par rapport au texte reçu. Montrons qu'elle peut être écrite de façon plus compacte en utilisant la notion de compréhension de liste ainsi que la méthode `join` qui sert à reconstruire un texte à partir des éléments de la liste construite par compréhension.

```
def keep_alnum_2(s):
    """Nettoie s en lui retirant les caractères non alphanumériques."""

    return "".join([i for i in s if i.isalnum()])
```

Analysez bien le fonctionnement de l'expression `"".join([i for i in s if i.isalnum()])`

5.6.3 Pratiquons encore : exercices UpyLaB 5.16 et suivant

AVEC OU SANS COMPRÉHENSION

Les exercices UpyLaB suivants peuvent aussi se résoudre sans utiliser la technique de compréhension de liste. Mais les solutions sont beaucoup plus courtes avec cette technique ! Nous vous demandons donc si possible de les réaliser avec du code utilisant des *compréhensions de liste*.

5.6.4 Exercice UpyLaB 5.16 (Vert)

Énoncé

Écrire une fonction `my_pow` qui prend comme paramètres un nombre entier `m` et un nombre flottant `b`, et qui renvoie une liste contenant les `m` premières puissances de `b`, c'est-à-dire une liste contenant les nombres allant de b^0 à b^{m-1}.

Si le type des paramètres n'est pas celui attendu, la fonction retournera la valeur `None`.

Exemple 1

L'appel suivant de la fonction :

```
my_pow(3, 5.0)
```

doit retourner :

```
[1.0, 5.0, 25.0]
```

Exemple 2

L'appel suivant de la fonction :

```
my_pow(3.0, 5.0)
```

doit retourner :

```
None
```

Exemple 3

L'appel suivant de la fonction :

```
my_pow('a', 'b')
```

doit retourner :

```
None
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(my_pow(2, 2.5))` par exemple.
- Pour contrôler le type des arguments, pensez à utiliser la fonction `type` et à comparer son résultat aux chaînes `int` ou `float`.
- Et n'oubliez pas de tester votre fonction avec des arguments de type incorrect. Cela ne doit pas entraîner d'erreur, mais la fonction retournera `None` dans ces cas.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 5.16*.

5.6.5 Exercice UpyLaB 5.17 (Bleu)

Énoncé

On considère une liste qui décrit une séquence `t`. Chaque élément de cette liste est un tuple de deux composantes : le nombre de répétitions successives de l'élément `x` dans la séquence `t`, et l'élément `x` lui-même.

Par exemple, la liste `[(1, 'He'), (2, 'l'), (1,'o')]` décrit la séquence `"Hello"`.

Écrire une fonction `decompresse` qui reçoit une telle liste en paramètre et renvoie la séquence `t` sous forme d'une nouvelle liste.

Exemple

L'appel suivant de la fonction :

```
decompresse([(4, 1), (0, 2), (2, 'test'), (3, 3), (1, 'bonjour')])
```

doit retourner :

```
[1, 1, 1, 1, 'test', 'test', 3, 3, 3, 'bonjour']
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Essayez d'utiliser une compréhension de liste pour cet exercice.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(decompresse([(1, 'He'), (2, 'l'), (1,'o')])` par exemple.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.17*.

5.6.6 Exercice UpyLaB 5.18 (Rouge)

Énoncé

Écrire une fonction `my_filter` qui reçoit une liste `lst` et une fonction booléenne `f` en paramètres et renvoie une nouvelle liste constituée des éléments de `lst` pour lesquels la fonction `f` renvoie `True`.

Exemples

Pour tester dans votre IDE (Thonny ou PyCharm par exemple) la fonction `my_filter`, vous allez devoir définir une fonction booléenne `f` et la passer en argument à la fonction `my_filter`.

Vous pourrez donc d'abord définir la fonction `f` à l'aide du mot-clé `def`, mais sachez que l'on peut aussi définir directement la fonction `f` lors de l'appel à la fonction `my_filter` en utilisant ce qu'on appelle une fonction lambda.

Il s'agit de fonctions anonymes que l'on peut utiliser au moment même.

Exemple d'une fonction lambda :

```
lambda x : isinstance(x, int)
```

Cette fonction testera si l'objet `x` qu'on lui passe est de type `int` et fait le même travail que la fonction `is_int` plus classiquement définie ci-dessous :

```
def is_int(x):
    return isinstance(x, int)
```

On pourra alors appeler la fonction `my_filter` des deux manières suivantes :

```
my_filter(lst, lambda x : isinstance(x, int))
my_filter(lst, is_int)
```

La seule différence, c'est que dans le deuxième cas, il faut avoir défini la fonction `is_int` au préalable (mais l'avantage, c'est qu'on pourrait la réutiliser dans la suite du code, contrairement à la fonction lambda).

Notez qu'UpyLaB utilisera ces fonctions lambda dans ses tests.

Exemple 1

L'appel suivant de la fonction :

```
my_filter(['hello', 666, 42, 'Thierry', 1.5], lambda x : isinstance(x, int))
```

doit retourner :

```
[666, 42]
```

Exemple 2

L'appel suivant de la fonction :

```
my_filter([-2, 0, 4, -5, -6], lambda x : x < 0)
```

doit retourner :

```
[-2, -5, -6]
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— On rappelle qu'une fonction booléenne est une fonction qui retourne `True` ou `False`.
— Essayez d'utiliser une compréhension de liste pour cet exercice.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `my_filter(['hello', 666, 42, 'Thierry', 1.5], lambda x : isinstance(x, int))` par exemple, ou en reprenant les tests d'UpyLaB.
— Si la notion de fonctions lambda vous paraît confuse, ne vous en inquiétez pas trop. C'est un point que nous n'abordons pas davantage dans ce cours d'introduction au langage Python.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.18*.

5.7 Testons nos connaissances

5.7.1 Un petit quiz pour consolider tout cela

MINI QUIZ SUR LA COMPRÉHENSION DE LISTE

Note : Voir le quiz de la section 5.7.1

COMMENTAIRE SUR LE QUIZ

Ce quiz montre la richesse de Python qui permet de faire un certain traitement de façons totalement différentes. Notons que si nous regardons chacun des codes, certains sont simples et clairs (et corrects) et d'autres plus compliqués et parfois même erronés. L'art de la programmation est de trouver le code que tous comprendront facilement : ici **simple et clair** sont les mots-clés.

Notons qu'au contraire, certains geeks font des concours du code le plus incompréhensible : voir par exemple avec le langage C : Concours international de code C obscur.

On s'amuse comme on peut !

5.8 Lisons et sauvons les textes

5.8.1 Manipulation de fichiers textes

CONNECTONS NOS CODES À DES FICHIERS (PARTIE 1)

Note : Voir la vidéo de la section 5.8.1 : Comment manipuler un fichier texte en lecture

CONTENU DE LA VIDÉO

L'ouverture, la lecture et la fermeture d'un fichier *texte* sont présentées.

Code sur la console réalisé dans la vidéo

```
>>> mon_fichier = open('words.txt')
>>> ko = open("pas_bon.txt")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'pas_bon.txt'
>>> mot = open("/Users/tmassart/Enseignement/SVN/tmassart/MOOC/Ressources_textuelles_et_livres/
↪mots.txt")
>>> mon_fichier.readline()
'aa\n'
>>> mon_fichier.readline()
'aah\n'
>>> mon_fichier.readline().strip()
'aahing'
>>> for lig in mon_fichier:
...     print(lig.strip())
...
aahs
aal
aalii
...
zymurgy
>>> mon_fichier.readline()
''
>>> mon_fichier.close()
>>> type(mot)
<class '_io.TextIOWrapper'>
>>> mot.readline()      # peut provoquer une erreur d'encodage !
'a\n'
>>> mot.readline()
'à\n'
>>> mot.close()
>>> mot = open("/Users/tmassart/Enseignement/SVN/tmassart/MOOC/Ressources_textuelles_et_livres/
↪mots.txt", encoding="utf-8")
```

La vidéo en bref

L'ouverture d'un fichier texte avec `open`, ainsi que les méthodes `readline()`, `strip()` et `close` sont présentées. À nouveau, référez-vous à l' *aide-mémoire* pour les options et un résumé de ce qu'elles font.

5.8.2 Manipulation de fichiers textes (suite)

CONNECTONS NOS CODES À DES FICHIERS (PARTIE 2)

Note : Voir la vidéo de la section 5.8.2 : Comment manipuler un fichier texte (suite)

CONTENU DE LA VIDÉO

La vidéo précédente illustre des manipulations en lecture et écriture de fichier texte par du code Python.

Codes sur la console réalisés dans la vidéo

```
>>> with open('mots.txt', encoding="utf-8") as mots:
...     for m in mots:
...         if len(m) > 24:
...             print(m.strip())
...
anticonstitutionnellement
constitutionnalisassions
constitutionnaliseraient
hospitalo-universitaires
oto-rhino-laryngologiste
oto-rhino-laryngologistes
```

```
>>> f = open("res.txt", encoding="utf-8")
>>> f.write("Bonjour")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
io.UnsupportedOperation: not writable
>>> f = open("res.txt", "w", encoding="utf-8")
>>> f.write("Bonjour")
7
>>> f.write(" ça va ?\n")
9
>>> f.write("ok\n")
3
>>> f.close()
>>> f = open("res.txt", "w", encoding="utf-8")
>>> f.write("perdu\n")
6
>>> f.close()
>>> f = open("res.txt", "w", encoding="utf-8")
>>> f.write(666)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: write() argument must be str, not int
```

```
f_in = input("fichier de données : ")
f_out = input("fichier de résultats : ")
with open(f_out, 'w', encoding='utf-8') as r:
    deja_vu = set({})
    with open(f_in, encoding='utf-8') as entree:
        for texte in entree:
            if texte not in deja_vu:
                deja_vu |= {texte}
                r.write(texte)
```

La vidéo en bref

Une meilleure façon d'ouvrir et de fermer un fichier texte est présentée, avec l'instruction `with`. À nouveau, référez-vous à l'*aide-mémoire* pour les options et un résumé de ce qu'elle foit.

Des exemples de *filtres*, qui ouvrent des fichiers textes en lecture ou écriture et affichent ou écrivent des résultats sur d'autres fichiers textes, illustrent nos propos.

Remarque : les ensembles seront formellement vus au module 6

5.8.3 Les fichiers textes (suite)

QUELQUES INFORMATIONS FORT UTILES

1) Les informations sauvées dans des fichiers qui continueront à exister après la fin de l'exécution du programme *Python*, sont appelées dans le jargon informatique *informations persistantes* par opposition aux données volatiles créées en mémoire vive lors de l'exécution du programme et qui disparaissent en cours ou en fin de cette exécution. Les données persistantes sont généralement sauvées sur un disque local ou distant (dans le *cloud*), sous forme soit de fichiers soit de bases de données.
2) Le fichier mots.txt, téléchargeable ici (que vous pouvez ensuite enregistrer sur votre ordinateur dans le répertoire du projet Python en cours), a été produit par Christophe Pallier à partir de données du dictionnaire Français-Gutenberg de Christophe Pythoud. Il contient 336531 mots français. C'est un fichier "plain text", c'est-à-dire que vous pouvez le lire dans n'importe quel éditeur de texte, mais également via Python. Il va nous permettre de jouer avec les mots.
3) Le fichier texte words.txt, téléchargeable ici (que vous pouvez ensuite enregistrer sur votre ordinateur dans le répertoire du projet Python en cours) contient 370099 mots d'anglais et a été obtenu de la page github.com ici
4) Lorsque l'on ouvre une console PyCharm, par défaut le répertoire courant correspond à celui du projet ouvert. Il peut être intéressant de définir un autre répertoire (directory en anglais) de travail que celui par défaut en particulier pour y manipuler des fichiers. Le module Python `os` peut être utile pour cela en permettant avec la fonction `chdir`, de changer le répertoire courant.

```
import os
os.chdir('Users/tmassart/Python/scripts')
```

5) Parfois nous parlerons de chemin au lieu de nom de fichier. Par exemple :

```
'/Users/tmassart/Python/scripts/le-livre-de-la-jungle.txt'
```

qui donne le nom du fichier avec son chemin complet depuis le répertoire *racine* (`'/'`) sur l'ordinateur. Ici, chemin et nom de fichier seront donc des synonymes.

MANIPULATIONS STANDARD DE FICHIERS TEXTE

La liste des méthodes standard sur les fichiers est la suivante :

Instruction	Effet
f=open("fichier") :	ouvre "fichier" en lecture
f=open("fichier","w") :	ouvre "fichier" en écriture
f=open("fichier","a") :	ouvre "fichier" en écriture en rajoutant après les données déjà présentes
f.read() :	retourne le contenu du fichier f
f.readline() :	lit une ligne
f.readlines() :	renvoie la liste des lignes de f
f.write(s) :	écrit la chaîne de caractères s dans le fichier f
f.close() :	ferme f

5.8.4 Des exemples de manipulation de fichiers

EXEMPLES QUI MANIPULENT DES FICHIERS TEXTES

À nouveau, manipulons ensemble les notions que nous venons d'introduire, ici les fichiers textes, pour nous familiariser avec celles-ci. Je vous propose ici trois petits problèmes à résoudre.

GADSBY ET LA DISPARITION

En 1939, Ernest Wright a publié une nouvelle de 50000 mots appelée Gadsby qui ne contient pas la lettre "e". Comme "e" est la lettre la plus commune en anglais (et dans d'autres langues y compris le français), ce n'était pas une tâche facile.

Notez que Georges Perec a fait le même exercice en français, dans son livre **« La disparition » (1969)** où il définit ce qui a disparu comme "un rond pas tout à fait clos, fini par un trait horizontal".

Problème

Écrivez un script qui n'affiche que les mots qui ne contiennent pas de "e" et calculez le pourcentage de ceux-ci par rapport à l'ensemble des mots du fichier `mots.txt`.

Proposition de solution

Ci-dessous une proposition de solution (n'oubliez pas de mettre le fichier `mots.txt` dans le répertoire « courant »).

```
with  open('mots.txt', encoding = "utf-8") as fichier:
    total = 0
    cnt = 0
    for line in fichier:
        total = total + 1
        if 'e' not in line :
            print(line.strip())
            cnt = cnt + 1

print('Pourcentage de mots sans e:', cnt / total * 100.0, "pourcents")
```

Fonction avec un nom de fichier en paramètre

Notez que le nom externe d'un fichier est une chaîne de caractères, comme "mot.txt" qui identifie un fichier dans le répertoire courant. Ainsi nous pouvons écrire une fonction, avec un nom de fichier reçu en paramètre : lors de chaque appel à la fonction, le paramètre formel sera donc une variable locale de type chaîne de caractères qui contiendra le nom du fichier concerné.

Par exemple, si nous voulons écrire une fonction booléenne `texte_sans_e` qui renvoie `True` si le texte contenu dans le fichier reçu en paramètre ne contient nulle part la lettre `"e"` (ni les lettres é, è, ê, ë), une solution est donnée ci-dessous.

Proposition de solution

```
def texte_sans_e(fichier_recu):
    """renvoie True si le texte contenu dans fichier_recu ne contient pas la lettre e,
    ni les lettres ``é``, ``è``, ``ê``, ``ë``
    Hypothèse: le fichier existe
    """
    with open(fichier_recu, encoding="utf-8") as fichier:
        texte = fichier.read()
        texte = texte.replace('é', 'e')
        texte = texte.replace('è', 'e')
        texte = texte.replace('ê', 'e')
        texte = texte.replace('ë', 'e')
    return 'e' not in texte
```

TRIPLE DOUBLES LETTRES CONSÉCUTIVES

Problème

Après avoir téléchargé le fichier `words.txt`, en cliquant words.txt ici, donnez la liste des mots anglais provenant du fichier `words.txt`, avec trois doubles lettres consécutives. Je vous donne un couple de mots qui sont presque candidats, mais pas tout à fait. Par exemple, le mot “committee” serait parfait s’il n’y avait pas ce “i” au milieu. Ou “Mississippi” : si on retirait les “i”, cela marcherait. Il y a cependant au moins un mot qui possède trois paires consécutives de lettres. C’est peut-être le seul mot, ou il peut y en avoir 500. Existe-il un mot avec les mêmes caractéristiques dans le fichier mots.txt ?

QUADRUPLE DOUBLES LETTRES NON CONSÉCUTIVES

Problème

Donnez la liste des mot français, provenant du fichier `mots.txt`, avec quatre doubles lettres éventuellement consécutives. Je vous donne un mot qui est presque candidat, mais pas tout à fait. Par exemple, le mot “commissionnaire” serait parfait s’il se terminait par « airre » ; il y a donc trois doubles lettres mais pas quatre. Existe-il un mot avec les mêmes caractéristiques dans le fichier `words.txt` ?

Proposition de solution

Ci-dessous une proposition de solution.

```
"""auteur: Thierry Massart
  date : 13 avril 2023
  Code qui joue avec les fichiers (locaux) 'words.txt' et 'mots.txt'
"""


def triple_double_lettres(mot):
    """renvoie vrai si le mot contient trois double lettres consécutives"""

    i = 0
    n = len(mot)
    res = False
    while not res and i < n - 5:
        res = mot[i] == mot[i + 1] and mot[i + 2] == mot[i + 3] \
              and mot[i + 4] == mot[i + 5]
        i += 1
    return res


def n_double_non_consecutifs(mot, n):
    """renvoie vrai si le mot contient n double lettres
       éventuellement non consécutives"""

    i = 0
    long = len(mot)
    cont = 0
    while cont < n and i < long - 1:
        if mot[i] == mot[i + 1]:
            cont += 1
            i += 2
        else:
            i += 1
    return cont == n


def quadruple_double_non_consecutifs(mot):
    """renvoie vrai si le mot contient quatre double lettres
```

(suite sur la page suivante)

(suite de la page précédente)

```
      éventuellement non consécutives"""

   return n_double_non_consecutifs(mot, 4)


# code principal
print("triple-doubles consécutifs avec words.txt")
with open('words.txt', encoding="UTF-8") as mots:
    for m in mots:
        mon_mot = m.strip()
        if triple_double_lettres(mon_mot):
            print(mon_mot)

print("triple-doubles consécutifs avec mots.txt")
with open('mots.txt', encoding="UTF-8") as mots:
    for m in mots:
        mon_mot = m.strip()
        if triple_double_lettres(mon_mot):
            print(mon_mot)

print("quadruples-doubles non consécutifs avec mots.txt")
with open('mots.txt', encoding="UTF-8") as mots:
    for m in mots:
        mon_mot = m.strip()
        if quadruple_double_non_consecutifs(mon_mot):
            print(mon_mot)

print("quadruples-doubles non consécutifs avec words.txt")
with open('words.txt', encoding="UTF-8") as mots:
    for m in mots:
        mon_mot = m.strip()
        if quadruple_double_non_consecutifs(mon_mot):
            print(mon_mot)
```

5.8.5 Les traitements de fichiers avec UpyLaB 5.19 et suivants

PRATIQUONS AVEC UPYLAB

Maintenant que vous avez bien assimilé les manipulations de fichiers textes, à vous d'en faire de façon autonome avec les exercices UpyLab suivants :

Exercice UpyLaB 5.19 (Vert)

Énoncé

D'après Wikipedia, un acrostiche est un poème, une strophe ou une série de strophes fondés sur une forme poétique consistant en ce que, lues verticalement de haut en bas, la première lettre ou, parfois, les premiers mots d'une suite de vers composent un mot ou une expression en lien avec le poème.

Écrire une fonction `acrostiche` qui reçoit en paramètre le nom d'un fichier et qui retourne la chaîne de caractères formée par les premières lettres de chaque ligne du fichier.

Exemple

Sachant que le fichier Apollinaire.txt contient le texte

```
Mon aimée adorée avant que je m'en aille
Avant que notre amour triste défaille
Râle et meure ô m'amie une fois
Il faut nous promener tous les deux seuls dans les bois
Alors je m'en irai plus heureux que les rois.
```

l'appel suivant de la fonction :

```
acrostiche("Apollinaire.txt")
```

doit retourner :

```
"MARIA"
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Les fichiers utilisés par UpyLaB pour tester la fonction sont accessibles aux adresses suivantes :
  - https://upylab.ulb.ac.be/pub/data/acro1.txt
  - https://upylab.ulb.ac.be/pub/data/acro2.txt
  - https://upylab.ulb.ac.be/pub/data/acro3.txt

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(acrostiche(("Apollinaire.txt")))` par exemple, en veillant à ce que le fichier utilisé soit bien présent dans le répertoire courant.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 5.19*.

5.8.6 Exercice UpyLaB 5.20 (Vert)

Énoncé

Écrire une fonction `nouveaux_heros` dont le but consiste à modifier les héros d'une histoire.

La fonction acceptera deux paramètres : * le premier sera une chaîne de caractères précisant le nom du fichier contenant l'histoire initiale ; * le deuxième sera une chaîne de caractères précisant le nom du fichier dans lequel sera sauvegardée l'histoire modifiée comme précisé ci-dessous.

Dans l'histoire initiale, présente dans le fichier dont le nom est donné en premier argument, trois protagonistes interviennent : Pierre, Paul et Jacqueline. La fonction devra remplacer ces trois héros par, respectivement, Paul, Tom et Mathilde. Le texte ainsi modifié sera alors stocké dans le fichier dont le nom est donné en deuxième argument. Aucune autre modification ne sera apportée au texte initial.

Exemple

Sachant que le fichier `upylab.ulb.ac.be//pub/data/histoire_1.txt` contient le texte :

```
Si Pierre est le fils de Paul, et si Paul est le frère de Jacqueline, qui est Pierre pour␣
↪Jacqueline ?
```

après l'appel suivant de la fonction :

```
nouveaux_heros("upylab.ulb.ac.be//pub/data/histoire_1.txt", "nouvelle_histoire_1.txt")
```

le fichier dont le nom est `nouvelle_histoire_1.txt` doit contenir :

```
Si Paul est le fils de Tom, et si Tom est le frère de Mathilde, qui est Paul pour Mathilde ?
```

Consignes

- Vous pouvez supposer que le fichier dont le nom est passé en premier argument existe bien et est au format UTF-8.
- Les fichiers utilisés par UpyLaB pour tester la fonction sont accessibles aux adresses suivantes :
  - https://upylab.ulb.ac.be/pub/data/histoire_1.txt
  - https://upylab.ulb.ac.be/pub/data/histoire_2.txt
- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre fonction, assurez-vous de copier le fichier à lire dans le répertoire courant (ou de passer comme argument le chemin complet vers le fichier).
- L'utilisation de la méthode `replace` sur les chaînes de caractères va s'avérer utile. Mais attention à Paul qui se trouve à la fois dans les anciens héros et les nouveaux. Une façon de procéder pourrait consister à remplacer, provisoirement, la chaîne `"Paul"` par une autre valeur (attention toutefois à ce que cette valeur provisoire ne soit pas elle-même présente dans le texte), ou, plus astucieusement, de réfléchir à l'ordre dans lequel apporter les modifications attendues.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 5.20*.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une solution à l'exercice : vidéo de résolution de l'exercice UpyLaB 5.20

5.8.7 Exercice UpyLaB 5.21 (Bleu)

Énoncé

Écrire une fonction `liste_des_mots` qui reçoit en paramètre le nom d'un fichier texte, que la fonction doit ouvrir, et qui renvoie la liste des mots contenus dans le fichier.

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- On peut supposer que le fichier est présent et dans le bon format en encodage UTF-8.
- Les mots dans la liste seront écrits en minuscule et triés dans l'ordre donné par la codification UTF-8 (en utilisant la méthode `sort` par exemple), les accents n'étant pas gérés de façon spécifique (« a » et « à » sont deux mots différents).
- Un même mot ne peut pas se trouver deux fois dans la liste.
- Dans le fichier source, les mots peuvent être séparés par des caractères blancs habituels (caractère espace, tabulation, passage à la ligne), ou par n'importe quel caractère parmi les suivants :
  - " » ? ! : ; . , * = ( ) 1 2 3 4 5 6 7 8 9 0
- Certains des fichiers utilisés par UpyLaB pour tester la fonction sont accessibles aux adresses suivantes :
  - https://upylab.ulb.ac.be/pub/data/Zola.txt
  - https://upylab.ulb.ac.be/pub/data/le-petit-prince.txt

Notez que le fichier `le-petit-prince.txt` est libre de droit d'auteur sauf en France (voir https://fr.wikipedia.org/wiki/Le_Petit_Prince). Si vous habitez en France, vous ne pouvez donc légalement le télécharger, mais aucun souci ailleurs dans le monde.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Pour tester votre fonction, assurez-vous de copier le fichier à lire dans le répertoire courant (ou de passer comme argument le chemin complet vers le fichier).
- Si rien ne marche : consultez la *FAQ sur UpyLaB 5.21*.

5.8.8 Exercice upylab 5.22 (Rouge)

Énoncé

Écrire une fonction `wc(nomFichier)` qui ouvre le fichier en question et renvoie un tuple de trois nombres :

- le nombre de caractères (y compris les caractères de retour à la ligne)
- le nombre de mots
- le nombre de lignes.

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Nous définissons ici un mot comme étant une chaîne de caractères alphanumériques, c'est-à-dire répondant `True` à la méthode `isalnum()`, et maximale, c'est-à-dire entourée d'espaces ou de séparateurs ou de caractères de fin de phrase.
- Les fichiers utilisés par UpyLaB pour tester la fonction sont accessibles aux adresses suivantes :
  - https://upylab.ulb.ac.be/pub/data/wc1.txt
  - https://upylab.ulb.ac.be/pub/data/Zola.txt
  - https://upylab.ulb.ac.be/pub/data/le-petit-prince.txt

  Pour information, le premier fichier contient les caractères `a2x!§t5\n` (`\n` désignant le caractère de fin de ligne). L'appel de la fonction sur ce fichier retourne donc le tuple `(8, 2, 1)`, les deux mots étant `a2x` et `t5`.
  Notez que le fichier `le-petit-prince.txt` est libre de droit d'auteur sauf en France (voir https://fr.wikipedia.org/wiki/Le_Petit_Prince). Si vous habitez en France, vous ne pouvez donc légalement le télécharger, mais aucun souci ailleurs dans le monde.

— Vous pourriez être tenté d'utiliser la méthode `split`. Ce n'est peut-être pas une très bonne idée, car la liste des séparateurs est ici très longue. Par exemple, le fichier pourrait contenir la chaîne `"a$€ αω $BOnJOur"`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Pour tester votre fonction, assurez-vous de copier le fichier à lire dans le répertoire courant (ou de passer comme argument le chemin complet vers le fichier).
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.22*.

5.9 Manipulons les matrices

5.9.1 Remarque importante sur l'initialisation des tableaux

INITIALISATION DE TABLEAUX : ATTENTION !

Commençons par un peu de vocabulaire. En programmation, une liste simple d'éléments est aussi appelée *vecteur* ou *tableau à une dimension*.

Par exemple, pour n = 4, nous avons vu en section 5.2, qu'initialiser un vecteur `vec` à n composantes à 0 pouvait se faire avec le code :

```
vec = [0] * n
```

Nous avons également vu au début de la présente section que la compréhension de liste était une technique puissante pour créer une nouvelle liste. Par exemple, le code :

```
vec = [0 for i in range(n)]
```

a le même effet que le code juste au dessus.

Un tableau à *deux dimensions*, appelé aussi *matrice*, peut être construit en Python sous forme de **liste de listes**.

Par exemple une matrice de trois lignes et quatre colonnes est initialisée avec le code Python :

```
matrix = [[1, 2,  3,  4],
          [5, 6,  7,  8],
          [9, 10, 11, 12]]
```

`matrix` sera donc une liste de listes de valeurs.

La valeur d'indice (i,j) sera manipulée par `maxtrix[i][j]`.

`matrix[i]` est la ième ligne de `matrix` (à partir de 0).

Pour une valeur n entière (n = 4 par exemple), nous parlons de matrice **n x n** (**n fois n**), pour un tableau à n lignes et n colonnes (c'est-à-dire chaque ligne a n éléments). Pour initialiser une matrice **n x n** à 0 nous devons prendre quelques précautions.

En effet, `matrix = [[0] * n] * n` ne donnera pas le résultat escompté, comme le montre le code Python Tutor suivant :

Note : Voir cours en ligne pour l'animation du code

Nous voyons que chaque sous-liste est en fait la même liste ; ainsi, si nous modifions un élément, par exemple `matrix[2][2] = 666`, chaque ligne de la matrice sera modifiée.

Une solution est d'utiliser le code :

```
matrix = [[0]*n for i in range(n)]
```

qui construit une liste distincte pour chaque ligne de `matrix`.

Nous aurons l'occasion de voir encore plusieurs exemples avec des compréhensions de listes dans les sections suivantes.

Notons pour le vocabulaire informatique, qu'un *tableau à 3 dimensions* est également appelé **cube** (pensez au cube de Rubik par exemple).

5.9.2 À présent exerçons-nous sur les matrices

DES MATRICES PARTOUT !

La notion de matrice est bien connue en mathématiques. Elle est également bien connue pour toute personne qui fait des jeux, que ce soit des mots-croisés ou des sudokus dans le journal ou des jeux de plateau comme le jeu de dames, d'échec ou le jeu de go, ou encore des jeux vidéo qui se déroulent sur une vue en deux dimensions d'une carte géographique ou d'un plan de bâtiment ou autre comme un plan à deux dimensions de labyrinthe.

Comme déjà mentionné dans la section précédente (5.9.1), une matrice est représentée en Python sous le forme d'une liste de listes, chaque sous-liste représentant les informations d'une ligne de la matrices.

Ce qui suit donne des exemples très intéressants de manipulation de matrices. On y parle de carré magique, de transformation et aussi du jeu de puissance quatre.

CARRÉS MAGIQUES À GOGO !

D'après Wikipedia : un carré magique d'ordre `n` est composé de n^2 entiers strictement positifs, écrits sous la forme d'un tableau carré. Ces nombres sont disposés de sorte que leurs sommes sur chaque rangée, sur chaque colonne et sur chaque diagonale principale, soient égales. Un carré magique normal est un cas particulier de carré magique, constitué de tous les nombres entiers de 1 à n^2, où `n` est l'ordre du carré.

Le carré magique normal non trivial (n > 1) le plus simple est celui pour `n` valant 3 ($n^2 = 9$) :

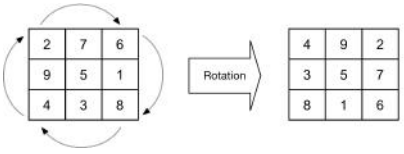
2	7	6
9	5	1
4	3	8

Carré magique 3x3

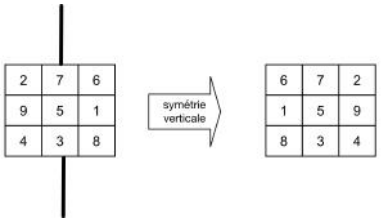
Le tableau Python correspondant peut être initialisé simplement

```
carre = [[2, 7, 6], [9, 5, 1], [4, 3, 8]]
```

Après avoir initialisé la matrice carre, nous pouvons construire l'ensemble des autres configurations qui constituent un carré magique normal avec `n = 3`. Pour cela, on peut faire des transformations de carre par **rotation**, par rapport à l'"**axe vertical** au milieu, ou à l'"**axe horizontal** au milieu ou à une des deux **diagonales**.



Rotation d'une matrice 3x3



Symétrie verticale d'une matrice 3x3

Dans ce but, pouvez-vous construire une fonction `rotation` qui reçoit un tableau `carre` de dimension `n x n` et qui renvoie un autre tableau correspondant à `carre` après une rotation de 90° vers la droite ? Pour avoir la valeur de `n`, il suffira de faire `n = len(carre)`. Nous pouvons supposer que chaque élément de `carre` est bien une liste de la même taille contenant des entiers.

Proposition de solution

Ci-dessous une proposition de solution.

```
def rotation(carre):
    """renvoie l'image de la matrice carre par rotation de 90° à droite"""

    n = len(carre)
    return [[carre[i][j] for i in range(n-1,-1,-1)] for j in range(n)]
```

SYMETRIE_VERTICALE

Ensuite, passons à la fonction `symetrie_verticale` qui reçoit également une telle matrice et renvoie une matrice où les valeurs sont changées par **symétrie verticale** par rapport au milieu comme expliqué.

Proposition de solution

Ci-dessous une proposition de solution.

```
def symetrie_verticale(carre):
    """renvoie l'image de la matrice carre par symétrie verticale"""

    n = len(carre)
    return [[carre[i][j] for j in range(n-1, -1, -1)] for i in range(n)]
```

NOTE

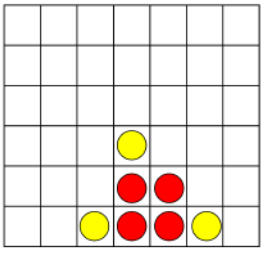
Notez que les deux fonctions sont de très beaux exemples d'utilisation de la technique de compréhension de liste.

JEUX DE PLATEAU ET AUTRES JEUX

Beaucoup de jeux de sociétés se jouent sur un plateau (jeu de dames, d'échec, ...). Le jeu de puissance 4 est un autre exemple : Puissance 4 (parfois aussi appelé 4 en ligne) est un jeu de stratégie.

Règles du jeu puissance 4 : le but du jeu est d'aligner une suite de 4 pions de même couleur sur une grille comptant 6 rangées et 7 colonnes. Chaque joueur dispose de 21 pions d'une couleur (par convention, en général jaune ou rouge). Tour à tour les deux joueurs placent un pion dans la colonne de leur choix, le pion coulisse alors jusqu'à la position la plus basse possible dans ladite colonne à la suite de quoi c'est à l'adversaire de jouer. Le vainqueur est le joueur qui réalise le premier un alignement (horizontal, vertical ou diagonal) consécutif d'au moins quatre pions de sa couleur. Si, alors que toutes les cases de la grille de jeu sont remplies, aucun des deux joueurs n'a réalisé un tel alignement, la partie est déclarée nulle.

L'état du jeu peut être représenté par une matrice de 6 lignes et 7 colonnes qui détermine pour chaque case si elle est occupée par un pion rouge, par un pion jaune ou si elle est libre. Un dernier petit élément pour donner l'état complet est de savoir qui doit jouer le prochain tour. Par exemple, le plateau de jeu suivant où les jaunes doivent jouer au prochain tour :



Plateau initial du jeu puissance 4

peut être représenté par la matrice :

```
jeu = [['V', 'V', 'J', 'R', 'R', 'J', 'V'],
       ['V', 'V', 'V', 'R', 'R', 'V', 'V'],
       ['V', 'V', 'V', 'J', 'V', 'V', 'V'],
       ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
       ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
       ['V', 'V', 'V', 'V', 'V', 'V', 'V']]
```

si l'on prend comme référence que le ligne du bas est la ligne 0 dans jeu (la première ligne du code Python) et que :

— "V" encode une case vide
— "R" encode une case avec un pion rouge
— "J" encode une case avec un pion jaune.

LECTURE D'UNE MATRICE D'ENTIERS

Un jeu sur un damier peut généralement être encodé avec une matrice de valeurs, par exemple entières. Ayant un fichier qui a un tel encodage, comment le *lire* et initialiser une matrice Python avec son encodage ?

Le code que nous vous avons proposé pour lire une liste d'entiers, peut être étendu pour lire une matrice d'entiers. Ici nous supposons que `fichier_encodage` est le nom du fichier à ouvrir et qui contient les valeurs, à raison d'une ligne par ligne de la matrice à initialiser où chaque entier est séparé par au moins une espace.

La fonction `lire_matrice` réalise une telle lecture et encodage et renvoie la matrice créée comme résultat :

```
def lire_matrice(fichier_encodage):
    with open(fichier_encodage, encoding='utf-8') as fichier_in:
        return [[int(colonne) for colonne in ligne.split()] for ligne in fichier_in]
```

Notons que l'utilisation de compréhensions de liste donne un code très court mais assez dense et difficile à comprendre pour un débutant.

5.9.3 Passons à la mise en pratique sur les matrices avec les exercices UpyLaB 5.26 et suivants

PASSONS À LA PRATIQUE AVEC UPYLAB

À nouveau, nous vous demandons, si vous le désirez, de réaliser les exercices UpyLaB 5.23 à 5.27 qui manipulent des matrices.

EXERCICE UPYLAB 5.23 (Vert)

Énoncé

Écrire une fonction `init_mat(m, n)` qui construit et renvoie une matrice d'entiers initialisée à la matrice nulle et de dimension `m x n`.

Exemple 1

L'appel suivant de la fonction :

```
init_mat(2, 3)
```

doit retourner :

```
[[0, 0, 0], [0, 0, 0]]
```

Exemple 2

L'appel suivant de la fonction :

```
init_mat(0, 0)
```

doit retourner :

```
[]
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(init_mat(2, 2))` par exemple.
- Si votre fonction semble retourner le bon résultat, mais n'est pourtant pas acceptée par UpyLaB, nous vous proposons de tester le code suivant dans votre IDE (après la définition de la fonction) :

  ```
  m = init_mat(3, 2)
  m[0][0] = 1 #modifie le premier élément de la matrice
  print(m)
  ```

  Que se passe-t-il ?
- N'hésitez pas à revoir la matière dispensée dans la section 5.9.1 concernant l'initialisation de tableaux, ni à utiliser Python Tutor et observer en particulier les diagrammes d'état.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 5.23*.

5.9.4 Exercice UpyLaB 5.24 (Vert)

Énoncé

Écrire une fonction `print_mat(M)` qui reçoit une matrice `M` en paramètre et affiche son contenu.

Les éléments d'une même ligne de la matrice seront affichés sur une même ligne, et séparés par une espace, les éléments de la ligne suivante étant affichés sur une nouvelle ligne.

Exemple

L'appel suivant de la fonction :

```
print_mat([[1, 2], [3, 4], [5, 6]])
```

doit afficher :

```
1 2
3 4
5 6
```

Consignes

— **Important** : Afin de permettre à UpyLaB de tester votre fonction, votre code contiendra également, après le code de la définition de la fonction `print_mat`, les instructions suivantes :

```
ma_matrice = eval(input())
print_mat(ma_matrice)
```

— Si la matrice vide `[]` est passée en argument, la fonction affiche une ligne vide.
— Dans cet exercice, la présence d'espaces en fin de ligne ne sera pas sanctionnée. Par contre, veillez à ce qu'il n'y ait pas de ligne supplémentaire.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Notez que la fonction `print_mat` ne retourne rien, mais procède à un affichage.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.24*.

5.9.5 Exercice UpyLaB 5.25 (Bleu)

Énoncé

Écrire une fonction `trace(M)` qui reçoit en paramètre une matrice *M* de taille $n \times n$ contenant des valeurs numériques (de type int ou float), et qui renvoie sa trace, c'est-à-dire la somme de tous les éléments de la première diagonale.

Exemple 1

L'appel suivant de la fonction :

```
trace([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

doit retourner :

```
15
```

En effet, 1 + 5 + 9 est égal à 15.

Exemple 2

L'appel suivant de la fonction :

```
trace([])
```

doit retourner :

```
0
```

Consignes

— Dans cet exercice, il vous est demandé d'écrire seulement la fonction `trace`. Le code que vous soumettez à UpyLaB doit donc comporter uniquement la définition de cette fonction, et ne fait en particulier aucun appel à `input` ou à `print`.
— Les éléments de la première diagonale sont les éléments dont l'indice de ligne est égal à l'indice de colonne. Ainsi :

$$trace(M) = \sum_{i=0}^{n-1} m_{ii}$$

— Vous pourrez supposer que les matrices passées en argument seront bien carrées (même nombre de lignes et de colonnes).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(trace([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))` par exemple.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.25*.

5.9.6 Exercice UpyLaB 5.26 (Bleu)

Énoncé

Une matrice $M = \{m_{ij}\}$ de taille $n \times n$ est dite *antisymétrique* lorsque, pour toute paire d'indices i, j, on a $m_{ij} = -m_{ji}$.

Écrire une fonction booléenne `antisymetrique(M)` qui teste si la matrice `M` reçue est antisymétrique.

Exemple 1

L'appel suivant de la fonction :

```
antisymetrique([[0, 1, 1], [-1, 0, 1], [-1, -1, 0]])
```

doit retourner :

```
True
```

Exemple 2

L'appel suivant de la fonction :

```
antisymetrique([[0, 1], [1, 0]])
```

doit retourner :

```
False
```

Exemple 3

L'appel suivant de la fonction :

```
antisymetrique([[1, -2], [2, 1]])
```

doit retourner :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que les matrices passées en argument sont bien carrées (même nombre de lignes et de colonnes).
— On rappelle qu'une fonction booléenne retourne les valeurs `True` ou `False`.
— La fonction retourne `True` pour la matrice vide.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(antisymetrique([[0, 1, 1], [-1, 0, 1], [-1, -1, 0]]))` par exemple.
— N'oubliez pas que la condition doit aussi s'appliquer lorsque les indices i et j sont égaux.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.26*.

5.9.7 Exercice UpyLaB 5.27 (Rouge)

Énoncé

Écrire une fonction `symetrie_horizontale(A)` qui reçoit une matrice carrée `A` (de taille $n \times n$) et qui renvoie l'image de `A` par symétrie horizontale par rapport à la ligne du milieu : la première ligne devenant la dernière, la seconde, l'avant-dernière, etc.

Exemple 1

L'appel suivant de la fonction :

```
symetrie_horizontale([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

doit retourner :

```
[[7, 8, 9], [4, 5, 6], [1, 2, 3]]
```

Exemple 2

L'appel suivant de la fonction :

```
symetrie_horizontale([['a', 'b'], ['c', 'd']])
```

doit retourner :

```
[['c', 'd'], ['a', 'b']]
```

Exemple 3

L'appel suivant de la fonction :

```
symetrie_horizontale([])
```

doit retourner :

```
[]
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que les matrices passées en argument sont bien carrées (même nombre de lignes et de colonnes).

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(symetrie_horizontale([[1, 2, 3], [4, 5, 6], [7, 8, 9]]))` par exemple.
— N'oubliez pas de tester le fonctionnement de votre fonction sur les cas limites, comme ici la matrice vide.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 5.27*.

5.10 Bilan du module

5.10.1 Qu'avons-nous vu dans ce module ?

BILAN DU MODULE 5

Note : Voir la vidéo de la section 5.10.1 : Bilan du module 5

RÉSUMÉ PLUS DÉTAILLÉ DE CE QUI A ÉTÉ VU DANS CE MODULE

Nous avons vu en section 1 :

— trois types de séquences Python : les chaînes de caractères (appelées aussi textes ou strings en anglais), les tuples et les listes ;
— comment créer une de ces séquences et l'assigner à une variable ;
— que pour une séquence `s`, la fonction prédéfinie `len(s)` donne sa longueur ;
— que la concaténation, par exemple `s + s`, et la répétition, par exemple `s * 10`, peuvent être utilisées ;
— la syntaxe à utiliser pour manipuler une composante `s[i]` pour une valeur `i` entre 0 compris et `len(s)` non compris ou avec des indices négatifs entre `-1` et `-len(s)` compris ;
— que l'instruction `for` permet de répéter un traitement sur chaque élément d'une séquence ;
— que la préposition `in` permet de tester si un élément est dans une séquence, et permet même pour les chaînes de caractères de tester si un sous-texte est dans un texte ;
— que les séquences peuvent aussi être comparées avec les opérateurs relationnels (<, <=, >, >=, ==, !=).

En section 2, nous avons vu :

— la notion de tranche (slicing), qui crée une copie d'une partie d'une séquence, la manière de modifier une composante ou d'étendre une liste (les tuples et chaînes de caractères étant immuables, c'est-à-dire non modifiables) ;
— l'identité d'un objet et le verbe `is` qui permet de déterminer si deux variables sont ou non des alias (deux noms différents pour le même objet) ;
— la possibilité d'avoir des listes et tuples non homogènes et imbriqués ;
— la notion d'emballage et de déballage (pack et unpack) entre les tuples et les variables qui reçoivent leur composantes.

En section 4, nous avons vu :

— le fonctionnement général de la plupart des fonctions prédéfinies et méthodes sur les trois types de séquences vues dans ce module. L'effet de la plupart est décrit dans l' aide mémoire que nous vous avons fourni.

En section 6, nous avons vu :

— la puissante notion de compréhension de liste ;
— en détail la notion de copie de listes avec les deux types de copie : superficielle (shallow) et profonde (deep).

En section 8, nous avons :

— expliqué comment manipuler un fichier texte soit en lecture soit en écriture ;
— vu que les fichiers sont des séquences de lignes de textes dont les éléments sont accessibles les uns après les autres ;
— montré comment ouvrir un tel fichier ;
— appris à lire son contenu ou écrire du texte dessus ;
— appris à le fermer ;
— fait référence à la partie de l' aide mémoire sur les fichiers, qui rappelle les instructions pour manipuler les fichiers.

Dans le module suivant nous ajoutons aux types de séquences que nous avons dans notre *boîte à outil* pour réaliser des programmes Python deux nouvelles classes de séquences : les *dictionnaires* et les *ensembles*. Nous verrons que les dictionnaires sont des structures de données particulièrement riches pour résoudre encore plus de problèmes intéressants.

CHAPITRE 6

Ensembles et dictionnaires : armes de construction massive

6.1 Manipulations de base des séquences ensembles et dictionnaires

6.1.1 Introduction aux ensembles et aux dictionnaires

LES ENSEMBLES

Note : Voir la vidéo de la section 6.1.1 : Les ensembles Python

CONTENU DE LA VIDÉO

La vidéo précédente présente le menu de ce module. Nous y présentons un nouveau type de structure de données : les *ensembles*.

Tests réalisés sur la console dans la vidéo

```
>>> personnages = {"rose", "serpent", "renard", "renard", "rose"}
>>> len(personnages)
3
>>> personnages
{'serpent', 'renard', 'rose'}
>>> "rose" in personnages
True
>>> "lion" in personnages
False
>>> for p in personnages:
...     print(p)
...
serpent
renard
rose
>>> s = set('abracadabra')
>>> t = set('alakazam')
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> s - t
{'b', 'r', 'c', 'd'}
>>> s | t
{'k', 'z', 'm', 'a', 'd', 'l', 'b', 'r', 'c'}
>>> s & t
{'a'}
>>> s ^ t
{'k', 'c', 'd', 'm', 'l', 'z', 'b', 'r'}
>>> s.add('x')
>>> s.remove('w')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'w'
>>> s.discard('w')
>>> s |= {'t'}
>>> s -= {'c', 'b', 'x'}
>>> s
{'a', 'd', 'r', 't'}
>>> s.remove('a')
>>> s
{'d', 'r', 't'}
>>> vide = set()
>>> d = {}
>>> type(d)
<class 'dict'>
>>> s.pop()
'd'
>>> s
{'r', 't'}
>>> s.pop()
'r'
>>> s.clear()
>>> s
set()
>>> t
{'k', 'm', 'a', 'l', 'z'}
>>> t[0]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'set' object does not support indexing
>>> a = {1, 2, 3}
>>> b = {1, 2, 3, 5}
>>> a < b
True
>>> a <= b
True
>>> a >= b
False
```

La vidéo en bref

Nous montrons comment manipuler les ensembles (set) Python, les façons de les créer, les opérateurs ensemblistes qui en créent d'autres, les méthodes d'ajout et de suppression d'éléments ainsi que les opérateurs d'inclusion stricte ou non dans les deux sens.

Référez-vous à l' aide mémoire pour avoir un résumé de ce que fait chaque méthode sur les ensembles.

LISTE DES MOTS D'UN FICHIER

Comme exemple de script qui utilise un ensemble Python, nous pouvons écrire une version plus complète et plus compacte du code proposé dans le module précédent, qui lit un fichier de mots (pour simplifier, nous supposons que chaque mot est séparé du suivant par une espace ou un passage à la ligne `'\n'`) et écrit sur un autre fichier la liste des mots rencontrés, un mot par ligne, en supprimant les doublons. De plus, cette version trie les mots.

```
with open('mots_un_par_ligne_avec_duplic.txt') as mots:
   mots_valides = set(mots.read().split()) # set permet de supprimer les doublons
liste_mots = list(mots_valides)            # transforme en liste
liste_mots.sort()                          # trie la liste

with open('liste_mots.txt', 'w') as res:   # écriture des mots
    for m in liste_mots:
        res.write(m + '\n')
```

6.1.2 Les dictionnaires

LES DICTIONNAIRES

Note : Voir la vidéo de la section 6.1.2 : Les dictionnaires Python

CONTENU DE LA VIDÉO

Dans la vidéo précédente nous introduisons maintenant les dictionnaires : un type de données très important étant donné les possibilités importantes qu'il ouvre en matière de programmation.

Tests réalisés dans la vidéo sur la console et avec un script

```
>>> ang2fr = {'one':'un', 'two': 'deux', 'three': 'trois'}
>>> type(ang2fr)
<class 'dict'>
>>> ang2fr['four'] = 'quatre'
>>> ang2fr
{'one': 'un', 'two': 'deux', 'three': 'trois', 'four': 'quatre'}
>>> ang2fr['two']
'deux'
>>> ang2fr['five']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'five'
>>> d = {}
>>> d = {'ent': 1, 'list': [1, 2, 3], (1,2): 3.14, dic: ang2fr}
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'dic' is not defined
>>> d = {'ent': 1, 'list': [1, 2, 3], 1:'un',  (1,2): 3.14, 'dic': ang2fr}
>>> d[[3,4]] = 45
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> d['dic']
{'one': 'un', 'two': 'deux', 'three': 'trois', 'four': 'quatre'}
>>> d['dic']['two']
'deux'
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> d['list'][-1]
3
>>> d = {'joe': 26, 'bob': 33, 'joe': 36}
>>> d
{'joe': 36, 'bob': 33}
>>> ang2fr
{'one': 'un', 'two': 'deux', 'three': 'trois', 'four': 'quatre'}
>>> len(ang2fr)
4
>>> 'one' in ang2fr
True
>>> 'deux' in ang2fr
False
>>> for cle in ang2fr:
...     print(cle)
...
one
two
three
four
>>> del(ang2fr['four'])
>>> ang2fr
{'one': 'un', 'two': 'deux', 'three': 'trois'}
>>> tel = {}
>>> tel['Baroud','Bill'] = '065-37-07-56'
>>> tel['II','Albert'] = '02-256-89-14'
>>> tel['Poelvoorde','Benoit'] = '081-23-89-65'
>>> tel
{('Baroud', 'Bill'): '065-37-07-56', ('II', 'Albert'): '02-256-89-14', ('Poelvoorde', 'Benoit'):
↪'081-23-89-65'}
>>> for last, first in tel:
...     print(first, last, tel[last, first])
...
Bill Baroud 065-37-07-56
Albert II 02-256-89-14
Benoit Poelvoorde 081-23-89-65
```

```
def histogram(s):
    """Renvoie le dictionnaire des lettres dans s avec leur fréquence"""
    d = {}
    for c in s:
        if c in d:
            d[c] += 1
        else:
            d[c] = 1
    return d

h = histogram('brontosaurus')
print(h)
```

La vidéo en bref

Nous montrons comment manipuler les dictionnaires (dict) Python, les façons de les créer avec pour chaque élément sa clé (key) et sa valeur (value), d'ajouter des nouveaux éléments, ainsi que la façon de manipuler les dictionnaires ou leurs éléments.

POURQUOI AVOIR DES TYPES NON MODIFIABLES / HASHABLES ?

Dans la vidéo précédente, nous avons vu, lorsque nous avons demandé à l'interpréteur d'ajouter à un dictionnaire une composante avec un clé de type liste, que l'opération n'avait pas eu lieu et que l'interpréteur avait donné le message d'erreur

```
TypeError: unhashable type: 'list'.
```

Mais qu'est-ce qu'un type « hashable » ?

Ne vous en faites pas, ce n'est pas un scénario de film d'horreur avec du sang partout !

Pour Python, une valeur de type « hashable » est un synonyme pour une valeur non modifiable. Nous avons vu que les objets simples de type entier (`int`), fractionnaire (`float`), booléen (`bool`), vus aux Modules 2 et 3 ainsi que les chaînes de caractères (`str`) et les `tuples`, vus au Module 5, sont non modifiables.

Ce sont donc tous des exemples de valeurs hashables. À côté de cela, nous avons vu, au module précédent et dans ce module, que les listes, les ensembles et les dictionnaires sont des types modifiables. En effet dans les trois types modifiables nous pouvons modifier des composantes ou en ajouter ou en supprimer.

L'utilité pratique des tuples par rapport aux listes apparaît donc quand on veut ajouter un élément dans un ensemble ou un dictionnaire. L'idée est que Python ne désire pas gérer des ensembles dont les **valeurs des éléments** pourraient changer en cours de route ; de même pour des clés de dictionnaires potentiellement modifiées durant l'exécution du programme.

INSTRUCTION FOR ET ENSEMBLE OU DICTIONNAIRE : ATTENTION

Python impose une restriction à l'utilisation de l'instruction `for` pour itérer tous les éléments d'un dictionnaire ou d'un ensemble : le corps du `for` ne peut ajouter ou supprimer de composante dans la séquence sur laquelle il itère. Par exemple le code suivant provoque une erreur à l'exécution :

```
s = {'a', 'b', 'c', 'd', 'e'}
for i in s:
    if i in 'aeiouy':
        s.remove(i)

Traceback (most recent call last):
  File "<input>", line 1, in <module>
RuntimeError: Set changed size during iteration
```

Cette restriction est assez sensée puisqu'il faut permettre à l'interpréteur, qui dans cet exemple au départ veut réaliser cinq itérations (une par élément de l'ensemble s), de s'y retrouver.

Note : Petite astuce pour les utilisateurs d'ordinateurs Mac : comment écrire le caractère « | » sur mon clavier ? Tapez `alt + Majuscule + L`

6.1.3 Petits exemples manipulant ensembles et dictionnaires

QUELQUES EXEMPLES

À nouveau pour assimiler la matière vue dans cette section, nous développons ensemble quelques exemples qui manipulent fichiers textes, ensembles et dictionnaires. Pour chaque problème, nous vous invitons à écrire une solution. Ensuite, comparez votre solution avec celle proposée.

LETTRES COMMUNES À DEUX MOTS

Problème

Dans la section 5 du module 5, nous avons déjà écrit un code pour trouver la liste des lettres communes à deux mots. Ceci se fait de façon immédiate en utilisant des ensembles avec le code :

```
list(set("pommeee") & set("banane")) #liste des lettres communes ['e']
```

LISTE DES MOTS CONTENANT LES 6 VOYELLES

Si l'envie nous prend.

Problème

Trouver et afficher la liste des mots du fichier «mots.txt » qui contiennent toutes les voyelles `('a', 'e', 'i', 'o', 'u', 'y')`.

La proposition de code suivante, qui utilise un ensemble, fera parfaitement l'affaire.

```
with open('mots.txt') as f:
    print([w.strip() for w in f if {'a','e','i','o','u','y'} <= set(w)])
```

HISTOGRAMME DE LA FRÉQUENCE DES LETTRES

Pour vous faciliter le travail, et pour que vous puissiez le relire et refaire cet exercice par vous-même, nous retranscrivons ci-dessous le code, présenté dans la capsule précédente, qui utilise un dictionnaire pour déterminer comment compter la fréquence de chaque lettre dans une chaîne de caractères. Cet exemple est en effet une belle illustration de l'utilisation de dictionnaire.

Problème

Écrivez la fonction `histogram(s)` qui reçoit une chaîne de caractère `s` et renvoie un dictionnaire dont les clés sont les lettres rencontrées et la valeur associée est la fréquence de la lettre dans `s`.

```
def histogram(s):
    """Renvoie le dictionnaire des lettres dans s avec leur fréquence."""
    d = {}   # dictionnaire vide
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
     return d
```

Ce code :

```
h = histogram('brontosaurus')
print(h)
```

renvoie :

```
{'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
```

LA LISTE DES AMIS

Supposons que nous ayons envie d'avoir une vue globale des amis de chacune de nos connaissances. Pour cela, nous pouvons construire un dictionnaire dont les clés sont les prénoms des personnes nommées (pour simplifier on suppose que deux personnes n'ont pas le même prénom) et la valeur de chaque entrée est l'ensemble des amis de la personne.

Problème

Écrivez une fonction qui reçoit une liste de couples (prénom1, prénom2) voulant dire que prénom1 déclare prénom2 comme étant son ami et qui construit le dictionnaire tel que décrit plus haut.

Une solution, et son code de test, pourrait être donnée par le code suivant :

```
def construction_dict_amis(liste_amis):
    """ construit et renvoie un dictionnaire de personnes qui ont chacun
    l'ensemble de leurs amis
    liste_amis : liste de couple (p1, p2) : p1 a comme ami p2"""
    amis = {}
    for prenom1, prenom2 in liste_amis:
        if prenom1 not in amis:
            amis[prenom1] = {prenom2}
        else:
            amis[prenom1].add(prenom2) # pas d'effet si déjà présent
        if prenom2 not in amis:
            amis[prenom2] = set() # nouvelle entrée prenom2
    return amis

t = 'Thierry'
m = 'Michelle'
s = 'Sébastien'
p = 'Pierre'
a = 'Ariane'
q = 'Quidam'
ma_liste = [(t, m), (t, s), (t, p), (t, a), (m, t), (m, a), (m, p),
            (p, a), (a, p), (s, t), (p, s), (s, q)]
print(construction_dict_amis(ma_liste))
```

Dans le code de la solution proposée, notez en particulier que, par souci de cohérence, si un couple est lu, par exemple : `('Thierry', 'Sébastien')`, le code s'assure que les deux entrées sont créées dans le dictionnaire (même si l'ensemble des amis de `'Sébastien'` est vide).

6.1.4 Mise en pratique des ensembles et dictionnaires avec les exercices UpyLaB 6.1 et suivants

Réalisez de façon autonome les exercices 6.1 à 6.7 d'UpyLaB. Ces exercices vont vous permettre de manipuler les ensembles et les dictionnaires.

EXERCICE UPYLAB 6.1 (Vert)

Énoncé

Après avoir longuement réfléchi et un peu visité notre monde, le Petit Prince décide de ne pas rentrer sur sa planète mais de s'installer dans les Cévennes pour profiter de la belle nature qu'on y trouve. Il y trouve une petite demeure pour y habiter, et plusieurs de ses amis veulent l'aider en lui proposant des meubles, des denrées, des livres ou d'autres choses qui pourraient l'intéresser pour aménager son nouveau domicile.

Nous vous proposons de l'aider.

Écrire une fonction `inventaire(offres, objets)` où :

— `offres` est un dictionnaire contenant, comme clés, les objets proposés par les amis du Petit Prince, et comme valeurs associées, le nom de l'ami proposant cet objet
— `objets` est une liste contenant tous les objets dont on a besoin le Petit Prince.

La fonction retourne l'ensemble des amis chez qui il lui faut se rendre pour sa récolte.

Exemple

L'appel suivant de la fonction :

```
inventaire({'lit': 'Antoine', 'bibliothèque': 'Sébastien',
            'chaise': 'Isabelle', 'livre "le vieil homme et la mer"': 'Ernest',
            'sac de bonbons': 'Thierry', 'smartphone': 'Ted', 'table': 'Sophie'},
            ['sac de bonbons', 'table', 'chaise', 'lit', 'livre "le vieil homme et la mer"'])
```

doit retourner :

```
{'Thierry', 'Sophie', 'Isabelle', 'Antoine', 'Ernest'}
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que les arguments passés à la fonction sont du bon type, et que les objets souhaités par le Petit Prince figurent bien parmi les objets proposés par ses amis.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.1*.

6.1.5 EXERCICE UPYLAB 6.2 (Vert)

Énoncé

(D'après une idée de Jacky Trinh le 19/02/2018)

Monsieur Germain est une personne très âgée. Il aimerait préparer une liste de courses à faire à l'avance. Ayant un budget assez serré, il voudrait que sa liste de courses soit dans ses capacités. Son seul petit souci est qu'il a une très mauvaise vue et n'arrive donc pas à voir le prix associé à chaque produit contenu dans le catalogue de courses.

Écrire une fonction `calcul_prix(produits, catalogue)` où :
— `produits` est un dictionnaire contenant, comme clés, les produits souhaités par Monsieur Germain et comme valeurs associées, la quantité désirée de chacun d'entre eux,
— `catalogue` est un dictionnaire contenant tous les produits du magasin avec leur prix associé.

La fonction retourne le montant total des achats de Monsieur Germain.

Exemple

L'appel suivant de la fonction :

```
calcul_prix({"brocoli":2, "mouchoirs":5, "bouteilles d'eau":6},
            {"brocoli":1.50, "bouteilles d'eau":1, "bière":2,
             "savon":2.50, "mouchoirs":0.80})
```

doit retourner :

```
13.0
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que les arguments passés à la fonction sont du bon type, et que les produits souhaités par Monsieur Germain figurent bien dans le catalogue du magasin.

6.1.6 Exercice UpyLaB 6.3 (Bleu)

Énoncé

(D'après une idée de Jacky Trinh le 24/02/2018)

Un jury doit attribuer le prix du « Codeur de l'année ».

Afin de récompenser les trois candidats ayant obtenu la meilleure moyenne, nous vous demandons **d'écrire une fonction** `top_3_candidats(moyennes)` qui reçoit un dictionnaire contenant comme clés les noms des candidats et comme valeurs la moyenne que chacun a obtenue.

Cette fonction doit retourner un tuple contenant les noms des trois meilleurs candidats, par ordre décroissant de leurs moyennes.

Exemple

L'appel suivant de la fonction :

```
top_3_candidats({'Candidat 7': 2, 'Candidat 2': 38, 'Candidat 6': 85,
                 'Candidat 1': 8, 'Candidat 3': 17, 'Candidat 5': 83,
                 'Candidat 4': 33})
```

doit retourner :

```
('Candidat 6', 'Candidat 5', 'Candidat 2')
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que l'argument passé à la fonction est du bon type.
— Vous pourrez supposer que les candidats ont des moyennes différentes, et qu'ils sont plus de trois.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.3*.

6.1.7 Exercice UpyLaB 6.4 (Bleu)

Énoncé

Lors de prises de notes, il nous arrive souvent de remplacer des mots par des abréviations (bonjour est remplacé par bjr par exemple).

Nous allons utiliser un dictionnaire qui associe à chacune de ces abréviations sa signification.

Écrire une fonction `substitue(message, abreviation)` qui renvoie une copie de la chaîne de caractères `message` dans laquelle les mots qui figurent parmi les clés du dictionnaire `abreviation` sont remplacés par leur signification (valeur).

Exemple

L'appel suivant de la fonction :

```
substitue('C. N. cpt 2 to inf', {'C.' : 'Chuck',
                                 'N.' : 'Norris',
                                 'cpt' : 'counted',
                                 '2' : 'two times',
                                 'inf' : 'infinity'})
```

doit retourner :

```
'Chuck Norris counted two times to infinity'
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Pour simplifier, on suppose que les mots de la chaîne `message` sont séparés par des espaces.
— Vous pourrez supposer que les arguments passés à la fonction sont du bon type.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.4*.

6.1.8 Exercice UpyLaB 6.5 (Vert)

L'exercice suivant correspond à un problème dont la solution a déjà été donnée.

Mais pourrez-vous le refaire sans allez voir la solution donnée ?

Notez que la résolution de cet exercice ne donne aucun point pour le cours !

Énoncé

Écrire une fonction `construction_dict_amis` qui reçoit une liste de couples `(prenom1, prenom2)` signifiant que `prenom1` déclare `prenom2` comme étant son ami.

La fonction construit et renvoie un dictionnaire dont les clés sont les prénoms des personnes nommées, et la valeur de chaque entrée est l'ensemble des amis de la personne.

Exemple

L'appel suivant de la fonction :

```
construction_dict_amis([('Quidam', 'Pierre'),
                        ('Thierry', 'Michelle'),
                        ('Thierry', 'Pierre')])
```

doit retourner :

```
{'Quidam' : {'Pierre'}, 'Pierre' : set(), 'Thierry' : {'Michelle', 'Pierre'}, 'Michelle' : set()}
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Si, dans la liste reçue, nous avons le couple `(prenom1, prenom2)`, cela n'induit pas que `prenom1` figure parmi les amis de `prenom2`. Si le couple `(prenom2, prenom1)` n'est pas dans cette liste, nous aurons une amitié à sens unique !
— Vous pourrez supposer que les arguments passés à la fonction sont du bon type.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.5*.

6.1.9 Exercice UpyLaB 6.6 (Rouge)

Énoncé

Écrire une fonction `symetrise_amis` qui reçoit un dictionnaire `d` d'amis où les clés sont des prénoms et les valeurs, des ensembles de prénoms représentant les amis de chacun.

Cette fonction modifie le dictionnaire `d` de sorte que si une clé `prenom1` contient `prenom2` dans l'ensemble de ses amis, l'inverse soit vrai aussi.

La fonction accepte un second paramètre `englobe`.

Si `englobe` est vrai, la fonction ajoutera les éléments nécessaires pour symétriser le dictionnaire `d`.

Sinon, la fonction enlèvera les éléments nécessaires pour symétriser `d`.

Exemple 1

L'exécution du code suivant :

```
d = {'Thierry': {'Michelle', 'Bernadette'},
     'Michelle': {'Thierry'},
     'Bernadette': set()}
symetrise_amis(d, True)
print(d)
```

doit afficher, à l'ordre près :

```
{'Thierry': {'Michelle', 'Bernadette'},
 'Michelle' : {'Thierry'},
 'Bernadette' : {'Thierry'}}
```

Exemple 2

L'exécution du code suivant :

```
d = {'Thierry': {'Michelle', 'Bernadette'},
     'Michelle': {'Thierry'},
     'Bernadette': set()}
symetrise_amis(d, False)
print(d)
```

doit afficher, à l'ordre près :

```
{'Thierry': {'Michelle'},
 'Michelle' : {'Thierry'},
 'Bernadette' : set()}
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Notez que les prénoms figurant dans les ensembles d'amis font eux-même partie des clés du dictionnaire `d`.
— Vous pourrez supposer que les arguments passés à la fonction sont du bon type.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Notez que la fonction `symetrise_amis` ne retourne rien mais modifie le dictionnaire qui lui est passé en argument.
— Notez aussi qu'il ne faut pas modifier le nombre d'éléments d'une séquence sur laquelle on est en train d'itérer.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.6*.

6.1.10 Exercice UpyLaB 6.7 (Vert)

Énoncé

Écrire une fonction `prime_odd_numbers(numbers)` qui reçoit une liste de nombres et qui renvoie un couple d'ensembles contenant respectivement les nombres premiers présents dans la liste et les nombres impairs.

Pour cela, nous vons demandons **d'écrire au préalable deux fonctions annexes** qui seront appelées dans le corps de la fonction `prime_odd_numbers` :
— la fonction `even` qui accepte un nombre entier en paramètre et renvoie l'ensemble des nombres naturels pairs qui lui sont inférieurs ou égaux
— la fonction `prime_numbers(max_nb)` qui accepte un nombre entier en paramètre et renvoie l'ensemble des nombres premiers qui lui sont inférieurs ou égaux.

Exemple 1

L'appel de la fonction suivant :

```
prime_odd_numbers([1, 2, 6, 5, 11, 9, 13, 14, 12, 15, 17, 18])
```

retourne, à l'ordre près, :

```
({2, 5, 11, 13, 17}, {1, 5, 11, 9, 13, 15, 17})
```

Exemple 2

L'appel de la fonction suivant :

```
prime_odd_numbers([1, 4, 6, 12, 9, 15, 16, 21, 18])
```

retourne, à l'ordre près, :

```
(set(), {1, 9, 15, 21})
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Cet exercice a pour unique but de vous faire manipuler les ensembles et les opérations ensemblistes. Et donc, malgré les apparences, il n'y a pas de contradiction entre le fait que l'on vous demande d'écrire la fonction `even` et le fait que la fonction `prime_odd_numbers` s'intéresse aux nombres impairs. C'est volontaire.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Prêtez bien attention au type des paramètres pour chacune des fonctions à écrire.
— Quel argument allez-vous passer aux fonctions `even` et `prime_numbers` lorsque vous lez appelez dans le code de la fonction `prime_odd_numbers` ?
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.7*.

6.2 Petit quiz sur les ensembles et les dictionnaires

6.2.1 Mini quiz sur les ensembles et les dictionnaires

PETIT QUIZ SUR LES ENSEMBLES ET LES DICTIONNAIRES

Voici un petit quiz qui récapitule les notions vues jusqu'à présent dans ce module sur les ensembles et les dictionnaires.

Note : Voir le quiz de la section 6.2.1

6.3 Méthodes de manipulation des dictionnaires

6.3.1 Présentons les méthodes des dictionnaires

MÉTHODES DES DICTIONNAIRES

Note : Voir la vidéo de la section 6.3.1 : Les méthodes des dictionnaires Python

---

CONTENU DE LA VIDÉO

La vidéo précédente présente les méthodes associées aux dictionnaires.

Tests réalisés sur la console dans la vidéo

```
>>> d = {}
>>> d = {'yes': 'oui', 'no': 'non'}
>>> d2 = d.copy()
>>> d2
```

(suite sur la page suivante)

(suite de la page précédente)

```
{'yes': 'oui', 'no': 'non'}
>>> d
{'yes': 'oui', 'no': 'non'}
>>> d2.clear()
>>> d2
{}
>>> d = {'yes': 'oui', 'no': 'non'}
>>> d2 = d.copy()
>>> d2
{'yes': 'oui', 'no': 'non'}
>>> d2['yes'] = 'OUI'
>>> d2
{'yes': 'OUI', 'no': 'non'}
>>> d2.clear()
>>> d2
{}
>>> from copy import deepcopy
>>> d2 = deepcopy(d)
>>> d2 = {}.fromkeys('abcde', 0)
>>> d
{'yes': 'oui', 'no': 'non'}
>>> d['a']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'a'
>>> d.get('a', 'mot inconnu')
'mot inconnu'
>>> d.setdefault('tree', 'arbre')
'arbre'
>>> d
{'yes': 'oui', 'no': 'non', 'tree': 'arbre'}
>>> 'yes' in d
True
```

```
>>> d = {'a': 1, 'b': 2}
>>> d.keys()
dict_keys(['a', 'b'])
>>> d.values()
dict_values([1, 2])
>>> d.items()
dict_items([('a', 1), ('b', 2)])
>>> for cle, valeur in d.items():
...     print(cle, valeur)
...
a 1
b 2
```

```
>>> eng2fr = {'one': 'un', 'two': 'deux', 'three': 'trois'}
>>> help(dict.pop)
Help on method_descriptor:

pop(...)
    D.pop(k[,d]) -> v, remove specified key and return the corresponding value.
    If key is not found, d is returned if given, otherwise KeyError is raised

>>> eng2fr.pop('three')
'trois'
>>> eng2fr
```

(suite sur la page suivante)

(suite de la page précédente)

```
{'one': 'un', 'two': 'deux'}
>>> eng2fr.pop('four')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'four'
>>> eng2fr.pop('four', 'inconnu')
'inconnu'
```

```
>>> d = {'a':0}
>>> d.update(zip('bcd', range(1, 4)))
>>> d
{'a': 0, 'b': 1, 'c': 2, 'd': 3}
```

```
>>> l = [('a', 1), ('b', 2)]
>>> dict(l)
{'a': 1, 'b': 2}
>>> dict(zip('abc', range(3)))
{'a': 0, 'b': 1, 'c': 2}
```

La vidéo en bref

Nous présentons les nombreuses méthodes associées aux dictionnaires : `copy`, `clear`, `fromkeys`, `get`, `setdefault`, `keys`, `values`, `items`, `pop`, `update`, ainsi que l'utilisation du `in`, `dict` et du `deepcopy`.

Référez-vous à l'aide mémoire pour avoir un résumé de ce que fait chaque méthode sur les dictionnaires.

NOTES IMPORTANTES

Quel est l'effet de la méthode que j'utilise ?

Dans ce module et le module précédent, vous avez appris un certain nombre de méthodes sur les séquences. Pour ne pas écrire de code erroné, il est en particulier essentiel de bien connaître l'effet d'une méthode : si l'objet est non modifiable (valeur simple, chaîne de caractères, tuple, ...), généralement, la méthode renvoie un résultat (différent de `None`) ; dans le cas des objets modifiables (`list`, `dict`, `set`, ...), il faut bien voir si l'effet de la méthode est :

— de modifier l'objet et la méthode renvoie `None`
— de renvoyer un résultat sans modifier l'objet
— à la fois de modifier l'objet et de renvoyer un résultat.

Objets itérables

les fonctions `range` et les méthodes `keys()`, `values()`, et `items()` sur les dictionnaires ne produisent pas des séquences, mais des objets itérables.

Le code suivant crée un tel objet et l'assigne à une variable (ici `val`) et demande d'itérer dessus.

```
>>> val = {'a':2, 'b':4}.values()
>>> for v in val:
...     print(v)
...
2
4
>>> print(val[0])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'dict_values' object is not subscriptable
>>> list_val = list({'a':2, 'b':4}.values())
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> print(list_val[0])
2
```

Ensuite, le code essaye d'accéder à l'indice `0` de `val` ce qui donne une erreur car l'objet ne peut être manipulé avec des indices, comme on pourrait le faire avec une séquence (`list`, `tuple` ou `str`).

Si l'on veut construire une séquence, par exemple une liste, avec un objet généré par `range`, (resp. `d.keys()`, `d.values()` ou `d.items()`), il faut explicitement transformer l'objet produit, avec `list(range(...))` (resp. `list(d.keys())`, `list(d.values())` ou `list(d.items())`), comme ici avec la variable `list_val`.

6.3.2 Petits exemples manipulant des méthodes de dictionnaires

QUELQUES EXEMPLES

À nouveau pour assimiler l'utilisation des différentes méthodes sur les dictionnaires que nous venons d'introduire, nous développons ensemble quelques exemples qui manipulent ces différentes méthodes. Comme toujours, pour chaque problème, nous vous invitons à écrire une solution. Ensuite, comparez votre solution avec celle proposée.

HISTOGRAM VERSION AMÉLIORÉE

Problème

Réécrivez la fonction `histogram()` (fonction qui renvoie le dictionnaire des lettres dans `s` avec leur fréquence) sans instruction conditionnelle explicite, en utilisant la méthode `get`.

Voici une proposition de solution.

```
def histogram(s):
    """Renvoie le dictionnaire des lettres dans s
       avec leur fréquence."""
    d = {}
    for c in s:
        d[c] = d.get(c,0) + 1
    return d
```

CLÉS AYANT UNE VALEUR DONNÉE

Problème

Supposons que nous voulons écrire une fonction `get_keys(d, value)` qui reçoit une valeur `value` et un dictionnaire `d`, et qui renvoie la liste des clés d'une valeur du dictionnaire donné.

Une idée est de parcourir le dictionnaire et rajouter à une liste les clés correspondant aux éléments ayant la bonne valeur. Notons que les clés d'un dictionnaire sont uniques, mais il peut y avoir une même valeur pour différentes clés.

Un code possible pour la fonction, et un code de test associé sont donnés ici :

```
def get_keys(d, value):
    """Renvoie la liste des clés de d ayant la valeur value."""
    t = []
    for k in d:
        if d[k] == value:
            t.append(k)
    return t

d = histogram('evenement')
```

(suite sur la page suivante)

(suite de la page précédente)

```
print(get_keys(d,4))   # imprime ['e']
print(get_keys(d,1))   # imprime ['v', 'm', 't']
```

DICTIONNAIRE INVERSÉ

Problème

Il est fréquent de vouloir inverser un dictionnaire. Par exemple ayant un dictionnaire `amis`, dont chaque clé est un prénom de personne, et la valeur associée, la liste de ses amis, on pourrait vouloir le dictionnaire avec les personnes comme clés, et comme valeurs associées la liste des personnes qui le connaissent. Ici prenons un exemple plus simple.

Problème plus simple

La fonction `histogram` est pratique mais nous pourrions avoir besoin d'obtenir ces informations de manière inversée.

Exemple : Au lieu de :

```
{'m': 1, 'n': 2, 'e': 4, 't': 1, 'v': 1},
```

nous aimerions avoir

```
{1 : ['m', 't', 'v'], 2 : ['n'], 4: ['e']}
```

qui montre que la liste des lettres présentes une seule fois dans le texte est `['m', 't', 'v'],...`

Le problème est donc de savoir comment inverser les clés et les valeurs d'un dictionnaire.

L'idée est de créer un nouveau dictionnaire qui contiendra des listes. Pour chaque valeur du dictionnaire de départ, nous ajoutons une paire dans le nouveau dictionnaire si celle-ci n'est pas présente, sinon, nous mettons à jour la paire.

Un code possible pour la fonction, et un code de test associé sont donnés ici :

```
def invert_dict(d):
    """Renvoie le dictionnaire inversé de d."""
    inv = {}
    for k in d:
        val = d[k]
        if val not in inv:
            inv[val] = [k]
        else:
            inv[val].append(k)
    return inv

d = histogram('evenement')
inv_d = invert_dict(d)
print(inv_d)    # imprime {1: ['m', 't', 'v'], 2: ['n'], 4: ['e']}
```

6.3.3 Exercice UpyLaB 6.8 (Vert)

Énoncé

Écrire une fonction `store_email(liste_mails)` qui reçoit en paramètre une liste d'adresses e-mail et qui renvoie un dictionnaire avec comme clés les domaines des adresses e-mail et comme valeurs les listes d'utilisateurs correspondantes, triées par ordre croissant (UTF-8).

Exemple

L'appel de la fonction suivant :

```
store_email(["ludo@prof.ur", "andre.colon@stud.ulb",
             "thierry@profs.ulb", "sébastien@prof.ur",
             "eric.ramzi@stud.ur", "bernard@profs.ulb",
             "jean@profs.ulb" ])
```

retourne le dictionnaire :

```
{ 'prof.ur' : ['ludo', 'sébastien']
  'stud.ulb' : ['andre.colon']
  'profs.ulb' : ['bernard', 'jean', 'thierry']
  'stud.ur' : ['eric.ramzi'] }
```

Consignes

Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.8*.

6.3.4 Exercice UpyLaB 6.9 (Bleu)

Énoncé

Écrire une fonction `compteur_lettres(texte)` qui renvoie un dictionnaire contenant **toutes** les lettres de l'alphabet associées à leur nombre d'apparition dans `texte`.

Exemple

L'appel de la fonction suivant :

```
compteur_lettres("Dessine-moi un mouton !")
```

retourne :

```
{'a': 0, 'b': 0, 'c': 0, 'd': 1, 'e': 2,
 'f': 0, 'g': 0, 'h': 0, 'i': 2, 'j': 0,
 'k': 0, 'l': 0, 'm': 2, 'n': 3, 'o': 3,
 'p': 0, 'q': 0, 'r': 0, 's': 2, 't': 1,
 'u': 2, 'v': 0, 'w': 0, 'x': 0, 'y': 0,
 'z': 0}
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Les clés du dictionnaire seront les lettres de l'alphabet en minuscule. Si le texte contient des majuscules, celles-ci seront comptabilisées comme la lettre minuscule correspondante.
— Le texte passé en paramètre ne comportera aucun caractère accentué.
— Les espaces et autres caractères de ponctuation doivent être ignorés.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.9*.

6.3.5 Exercice UpyLaB 6.10 (Rouge)

Énoncé

Écrire une fonction `valeurs(dico)` qui doit fournir, à partir du dictionnaire donné en paramètre, une liste des valeurs du dictionnaire triées selon leur clé.

Exemple

L'appel de la fonction suivant :

```
valeurs({'three': 'trois', 'two': 'deux', 'one': 'un'})
```

retourne :

```
['un', 'trois', 'deux']
```

En effet, les clés correspondantes sont "one" < "three" < "two" (ordre UTF-8).

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que les clés du dictionnaire sont des chaînes de caractères.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— La méthode `sort` pourra être appliquée à la liste de clés.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.10*.

6.4 Quiz de synthèse de la matière

6.4.1 Quiz de synthèse sur les structures de données

QUIZ DE SYNTHESE

Note : Voir le quiz de la section 6.4.1

NOTE SUR L'ITÉRATION SUR LES FONCTIONS

Supposons comme dans le dernier quiz que l'on désire tester différentes fonctions avec les mêmes arguments en nombre et valeurs, Python permet d'itérer sur une séquence de fonctions comme montré dans l'exemple qui suit :

```
for f in (anagramme_str1, anagramme_str2, anagramme_0, anagramme_2,
          anagramme_3, anagramme_dico, anagramme_dico2,
          anagramme_list):
    print("avec la fonction", f.__name__)
    print(f('bonjour', 'jourbon'))
    print(f('bonjour', 'jouxron'))
    print(f('bonjour', 'bonjur'))
    print(f('bonjour', 'jjourbon'))
```

où les différentes fonctions `anagramme_xxx` ont toutes été définies avant. À chaque itération `f` référence une des fonctions de la séquence. Notez l'utilisation de l'attribut `__name__` pour imprimer le nom de la fonction utilisée à chaque itération.

6.5 Exercices de synthèse

6.5.1 Des exercices UpyLaB qui mobilisent plusieurs matières

EXERCICES DIVERS

Nous arrivons doucement à la fin de ce cours en ligne. Les exercices UpyLaB suivants peuvent parfois manipuler des dictionnaires parfois des ensembles, listes ou autres structures de données. C'est l'occasion pour vous de vous entraîner en récapitulant les matières vues jusqu'ici dans l'ensemble des modules du cours.

6.5.2 Exercice UpyLaB 6.11 (Vert)

Énoncé

Le Petit Prince a peur des baobabs qui, en poussant, pourraient abîmer sa minuscule planète. Par contre, il adore les arbres à fleurs que Monsieur Jardinier lui montre.

Touché par son enthousiasme, Monsieur Jardinier lui offre 1 000 graines de ces arbres à fleurs, mais maintenant, le Petit Prince doit trouver une planète suffisamment grande pour les accueillir, sachant que chacun de ces arbres nécessite une surface de 50 m2 pour se développer sereinement.

Écrire une fonction booléenne `bonne_planete(diametre)` qui reçoit en paramètre un nombre représentant le diamètre, en mètres, d'une planète candidate. La fonction retourne la valeur `True` ou `False` selon la planète convient ou non.

Écrire ensuite un programme qui lit un diamètre depuis l'entrée et affiche
- la chaîne de caractères `"Bonne planète"` si le résultat de l'appel à la fonction `bonne_planete` est `True`
- la chaîne de caractères `"Trop petite"` sinon.

Exemple 1

L'appel de la fonction suivant :

```
bonne_planete(500)
```

retourne :

```
"Bonne planète"
```

Exemple 2

L'appel de la fonction suivant :

```
bonne_planete(100)
```

retourne :

```
"Trop petite"
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Nous rappelons que l'aire d'une sphère, solide auquel est assimilée une planète pour les besoins de l'exercice, s'obtient par le calcul πd^2 où d désigne le diamètre de la sphère.

6.5.3 Exercice UpyLaB 6.12 (Vert)

Énoncé

Écrire une fonction `belongs_to_file(word, filename)` qui reçoit deux chaînes de caractères en paramètre. La première correspond à un mot, et la deuxième au nom d'un fichier contenant une liste de mots, chacun sur sa propre ligne. La fonction vérifie si le mot figure dans cette liste, et retourne `True` si c'est bien le cas, `False` sinon.

Exemple

L'appel de la fonction suivant, où `words.txt` est le fichier indiqué dans les consignes ci-dessous et en supposant qu'il se trouve dans le même répertoire que le programme :

```
belongs_to_file("renard", "words.txt")
```

retourne :

```
False
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Vous pourrez supposer que le fichier passé en paramètre contient bien une liste de mots, chacun sur sa propre ligne.
- N'oubliez pas d'ouvrir le fichier dans le code de la fonction.
- Le fichier utilisé par UpyLaB pour effectuer les tests est disponible à l'adresse `https://upylab.ulb.ac.be/pub/data/words.txt`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- N'oubliez pas d'ôter le caractère de fin de ligne lorsque vous voulez comparer les deux chaînes de caractères.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 6.12*.

6.5.4 Exercice UpyLaB 6.13 (Vert)

Énoncé

Voici le début d'une suite logique inventée par John Horton Conway (et connue donc sous le nom de suite de Conway).

```
1
1 1
2 1
1 2 1 1
1 1 1 2 2 1
3 1 2 2 1 1
...
```

Chaque ligne, à partir de la deuxième, décrit la précédente :

- la première ligne, 1, est formée de un "1", d'où la deuxième ligne : 1 1 ;
- la troisième ligne décrit la deuxième ligne, où l'on voit deux "1", d'où 2 1 ;

— la quatrième ligne décrit la troisième ligne, où l'on voit un "2" et un "1", d'où 1 2 1 1 ;
— et ainsi de suite.

Écrire une fonction `next_line(line)` qui reçoit une liste d'entiers, et qui retourne la liste correspondant à la ligne suivante.

Notez que les valeurs de la liste reçue sont toujours entières, mais cette dernière peut ne pas correspondre à une suite de Conway (par exemple [4,2] pourrait être donné).

Exemple 1

L'appel suivant de la fonction :

```
next_line([1, 2, 1, 1])
```

doit retourner :

```
[1, 1, 1, 2, 2, 1]
```

Exemple 2

L'appel suivant de la fonction :

```
next_line([1])
```

doit retourner :

```
[1, 1]
```

Exemple 3

L'appel suivant de la fonction :

```
next_line([])
```

doit retourner :

```
[1]
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Notez que l'appel `next_line([])` sur la liste vide retourne par convention la liste `[1]`.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Il n'est demandé ici que de définir la fonction. Mais pour tester son fonctionnement dans votre IDE, pensez à ajouter du code qui l'appelle et teste son résultat, sur plusieurs valeurs, comme `print(next_line([1, 2, 1, 1]))` par exemple.

N'hésitez pas à tester votre fonction sur les cas « limites » comme la liste vide, `[]`, ou la liste singleton `[1]`, en vous référant aux exemples donnés.

— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.13*.

Une solution possible

Un webinaire a été fait lors de la session 3, présentant une approche possible à l'exercice : vidéo de résolution de l'exercice UpyLaB 6.13

6.5.5 Exercice UpyLaB 6.14 (Vert)

Énoncé

(D'après une idée de Jacky Trinh le 15/02/2018)

Puissance 4 est un jeu de stratégie combinatoire abstrait dont le but est d'aligner une suite de quatre pions de même couleur sur une grille comptant six rangées et sept colonnes.

Chaque joueur possède vingt-et-un pions d'une couleur (jaune ou rouge). À chaque tour, le joueur suivant doit placer un pion dans la colonne de son choix. Le pion tombe dans la position la plus basse possible.

Le vainqueur est le premier qui a réussi à aligner horizontalement, verticalement ou diagonalement, de manière consécutive, quatre de ses pions.

Écrire une fonction `placer_pion(couleur, colonne, grille)` où :

— `couleur` est la couleur du pion, qui peut être soit `'R'` (rouge), soit `'J'` (jaune),

— `colonne` est l'indice de la colonne où le joueur souhaite placer le pion (allant de 0 à 6),

— `grille` est la grille de jeu sous forme de matrice.

Cette matrice contient donc six listes de lignes contenant chacune sept éléments.

La ligne à l'indice 0 représente le bas du jeu.

Les éléments de cette matrice seront soit `'R'`, soit `'J'` ou soit `'V'` pour un emplacement encore vide.

Cette fonction placera un pion dans la grille du jeu et renverra un couple de valeurs :

— si le placement est possible, la valeur booléenne `True` et la grille modifiée,

— sinon, la valeur booléenne `False` et la grille non modifiée.

Exemple 1

L'appel suivant de la fonction :

```
placer_pion("R", 3, [['V', 'V', 'V', 'J', 'V', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'V', 'V', 'V']])
```

doit retourner :

```
(True, [['V', 'V', 'V', 'J', 'V', 'V', 'V'],
        ['V', 'V', 'V', 'R', 'V', 'V', 'V'],
        ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
        ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
        ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
        ['V', 'V', 'V', 'V', 'V', 'V', 'V']])
```

Exemple 2

L'appel suivant de la fonction :

```
placer_pion("J", 4, [['J', 'J', 'J', 'J', 'R', 'R', 'R'],
                     ['R', 'R', 'R', 'R', 'R', 'V', 'V'],
                     ['J', 'J', 'J', 'J', 'J', 'V', 'V'],
                     ['V', 'R', 'J', 'R', 'J', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'J', 'V', 'V'],
                     ['V', 'V', 'V', 'V', 'R', 'V', 'V']])
```

doit retourner :

```
(False, [['J', 'J', 'J', 'J', 'R', 'R', 'R'],
         ['R', 'R', 'R', 'R', 'R', 'V', 'V'],
         ['J', 'J', 'J', 'J', 'J', 'V', 'V'],
         ['V', 'R', 'J', 'R', 'J', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'J', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'R', 'V', 'V']])
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que les arguments passés à la fonction seront valides.
— Il n'est pas demandé de vérifier si la grille contient un alignement gagnant.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.14*.

6.5.6 Exercice UpyLaB 6.15 (Rouge)

Énoncé

(D'après une idée de Jacky Trinh le 15/02/2018)

Puissance 4 est un jeu de stratégie combinatoire abstrait dont le but est d'aligner une suite de quatre pions de même couleur sur une grille comptant six rangées et sept colonnes.

Chaque joueur possède vingt-et-un pions d'une couleur (jaune ou rouge). À chaque tour, le joueur suivant doit placer un pion dans la colonne de son choix. Le pion tombe dans la position la plus basse possible.

Le vainqueur est le premier qui a réussi à aligner horizontalement, verticalement ou diagonalement, de manière consécutive, quatre de ses pions.

Écrire une fonction `gagnant(grille)` où `grille` est la grille de jeu sous forme de matrice.

Cette matrice contient donc six listes de lignes contenant chacune sept éléments.

La ligne à l'indice 0 représente le bas du jeu.

Les éléments de cette matrice seront soit `'R'`, soit `'J'` ou soit `'V'` pour un emplacement encore vide.

Cette fonction renverra `'R'` si le joueur à la couleur rouge a gagné, `'J'` si le joueur à la couleur jaune a gagné ou bien `None` si aucun joueur n'a gagné.

Exemple 1

L'appel suivant de la fonction :

```
gagnant([['V', 'V', 'V', 'J', 'R', 'R', 'J'],
         ['V', 'V', 'V', 'R', 'J', 'R', 'R'],
         ['V', 'V', 'V', 'V', 'R', 'J', 'J'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'J'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V']])
```

doit retourner :

```
'J'
```

Exemple 2

L'appel suivant de la fonction :

```
gagnant([['V', 'R', 'J', 'J', 'J', 'R', 'V'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V'],
         ['V', 'V', 'V', 'V', 'V', 'V', 'V']])
```

doit retourner :

```
None
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que la matrice passée à la fonction est valide, et qu'il n'y a pas à la fois une configuration gagnante pour chacune des couleurs.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.15*.

6.5.7 Exercice UpyLaB 6.16 (Rouge)

Énoncé

— **Écrire une fonction** `file_histogram(fileName)` qui prend en paramètre le nom, sous forme d'une chaîne de caractères, d'un fichier texte, et qui renvoie un dictionnaire associant à chaque caractère du texte contenu dans ce fichier son nombre d'occurrences.

— **Écrire une fonction** `words_by_length(fileName)` qui prend en paramètre le nom, sous forme d'une chaîne de caractères, d'un fichier texte, et qui renvoie un dictionnaire associant à une longueur l la liste triée (dans l'ordre utf-8 croissant) des mots de longueur l présents dans le texte contenu dans le fichier. Ces mots seront écrits en minuscules.

Exemple 1

L'appel de la fonction suivant :

```
file_histogram('/pub/data/Zola.txt')
```

retourne (à l'ordre près) :

```
{'f': 14, 'î': 1, "'": 10, 'v': 15, 'm': 24,
 'N': 1, 'o': 49, 'a': 95, 'h': 13, 'd': 37,
 'é': 21, 'b': 12, 'E': 2, '\n': 1, 'x': 2,
 'y': 3, 'U': 1, 'M': 2, 'u': 50, 'à': 3,
 'e': 168, 'i': 73, ' ': 209, 'r': 64, 'p': 24,
 'D': 3, 's': 105, ',': 35, 't': 82, '-': 1,
 'è': 4, 'g': 16, 'c': 39, '.': 10, '?': 1,
 'L': 1, 'n': 81, 'ç': 1, 'j': 4, 'A': 2,
 'l': 71, 'ô': 1, 'q': 5}
```

Exemple 2

L'appel de la fonction suivant :

```
words_by_length('/pub/data/Zola.txt')
```

retourne (à l'ordre près) :

```
{1: ['a', 'c', 'd', 'l', 'n', 's', 'à'],
 2: ['ce', 'de', 'du', 'en', 'et', 'il',
     'la', 'le', 'là', 'sa', 'se', 'si', 'un'],
 3: ['aux', 'des', 'ils', 'les', 'par', 'pas',
     'que', 'qui', 'ses', 'sol', 'une', 'vie'],
 4: ['avec', 'blés', 'ciel', 'dans', 'dont',
     'loin', 'plus', 'pour', 'sous', 'sève',
     'tous', 'voix'],
 5: ['armée', 'astre', 'avril', 'bruit', 'cette',
     'comme', 'coups', 'faire', 'flanc', 'futur',
     'grand', 'haies', 'noire', 'parts', 'pieds',
```

(suite sur la page suivante)

(suite de la page précédente)

```
    'pièce', 'plein', 'terre', 'vives', 'était'],
 6: ['allait', 'arbres', 'autres', 'baiser', 'besoin',
    'cassée', 'champs', 'chaque', 'droite', 'encore',
    'gauche', 'germes', 'gloire', 'grosse', 'herbes',
    'hommes', 'jeunes', 'plaine', 'rauque', 'rayons',
    'rumeur', 'siècle', 'soleil', 'suivre', 'toutes',
    'vertes', 'échine'],
 7: ['bientôt', 'chaleur', 'coulait', 'croyait',
    'fussent', 'germait', 'graines', 'lumière',
    'maheude', 'matinée', 'montait', 'poussée',
    'sillons', 'souffle', 'éclater', 'étaient'],
 8: ['campagne', 'enjambée', 'feuilles', 'jeunesse',
    'obstinés', 'profonds', 'récoltes', 'tapaient', 'épandait'],
 9: ['bourgeons', 'camarades', 'crevaient', 'enfantait', 'enflammés',
    'entendait', 'gerçaient', 'lentement', 'rayonnait'],
 10: ['accompagné', 'betteraves', 'gonflaient', 'maintenant',
     'nourricier', 'poussaient', 'rapprochés', 'rivelaines',
     'ronflement', 'vengeresse', 'échauffant'],
 11: ['débordement', 'germination', 'grandissant', 'jaillissait',
     'reconnaître', 'travaillées', 'ventilateur'],
 12: ['allongeaient', 'chuchotantes', 'continuaient'],
 13: ['distinctement'], 14: ['tressaillaient']}
```

Consignes

- Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
- Les fichiers utilisés par UpyLaB pour tester votre code sont :
  - https://upylab.ulb.ac.be/pub/data/words.txt
  - https://upylab.ulb.ac.be/pub/data/le-petit-prince.txt
  - https://upylab.ulb.ac.be/pub/data/Zola.txt
- Pour la fonction `words_by_length`, on supposera qu'un mot est une séquence de caractères alphabétiques. La méthode `isalpha()` pourra être utile.

Par exemple, la chaîne de caractères `'cat4dog'` sera considérée comme formant deux mots : `'cat'` et `'dog'`.

Veillez aussi à ce qu'un même mot n'apparaisse pas plusieurs fois dans la même liste.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Vérifiez que vos fonctions répondent bien à toutes les contraintes de l'énoncé.
- Veillez à ne pas faire trop de passages sur le contenu du fichier, qui peut être de taille conséquente. Sinon, UpyLaB pourrait stopper l'exécution de votre code qu'il trouve trop longue.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 6.16*.

6.5.8 Exercice UpyLaB 6.17 (y compris 6.17a, 6.17b et 6.17c) (Bleu)

Notez que cet exercice est découpé en quatre : l'exercice complet et 3 parties (UpyLaB 6.17a, 6.17b, 6.17c, 6.17). Chacun des 4 exercices est comptabilisé pour 1 point (soit 4 points au total).

Énoncé

Le sudoku est un jeu de logique dans lequel le joueur reçoit une grille de 9 x 9 cases, chacune pouvant contenir un chiffre de 1 à 9 ou bien être vide. La grille est divisée en 9 lignes, 9 colonnes ainsi qu'en 9 « sous-grilles », appelées régions, formées de 3 x 3 boîtes contiguës. Le but du jeu est de remplir les cases vides avec des chiffres de 1 à 9, de telle sorte que, dans chaque ligne, chaque colonne et chaque région, soient présents tous les chiffres de 1 à 9, sans doublons.

Écrire une fonction `check_sudoku` capable de vérifier si la grille passée en paramètre, sous forme d'une matrice 9 x 9 d'entiers, est une solution au problème du sudoku. La fonction retournera la réponse (`True` ou `False`).

Cette fonction devra utiliser les trois fonctions suivantes (que vous devez aussi définir) :

— `check_rows` qui prend en paramètre une grille sous forme de matrice à deux dimensions et vérifie si toutes les lignes sont valides (c'est-à-dire que sur chaque ligne, chaque chiffre apparaît une et une seule fois).
— `check_cols` qui prend en paramètre une grille sous forme de matrice à deux dimensions et vérifie si toutes les colonnes sont valides (c'est-à-dire que sur chaque colonne, chaque chiffre apparaît une et une seule fois).
— `check_regions` qui prend en paramètre une grille sous forme de matrice à deux dimensions et vérifie si toutes les régions sont valides (c'est-à-dire que dans chaque région, chaque chiffre apparaît une et une seule fois).

Exemple 1

L'appel de la fonction suivant :

```
check_sudoku([[9, 6, 3, 1, 7, 4, 2, 5, 8],
              [1, 7, 8, 3, 2, 5, 6, 4, 9],
              [2, 5, 4, 6, 8, 9, 7, 3, 1],
              [8, 2, 1, 4, 3, 7, 5, 9, 6],
              [4, 9, 6, 8, 5, 2, 3, 1, 7],
              [7, 3, 5, 9, 6, 1, 8, 2, 4],
              [5, 8, 9, 7, 1, 3, 4, 6, 2],
              [3, 1, 7, 2, 4, 6, 9, 8, 5],
              [6, 4, 2, 5, 9, 8, 1, 7, 3]])
```

retourne :

```
True
```

Exemple 2

L'appel de la fonction suivant :

```
check_sudoku([[1, 2, 3, 4, 5, 6, 7, 8, 9],
              [2, 3, 4, 5, 6, 7, 8, 9, 1],
              [3, 4, 5, 6, 7, 8, 9, 1, 2],
              [4, 5, 6, 7, 8, 9, 1, 2, 3],
              [5, 6, 7, 8, 9, 1, 2, 3, 4],
              [6, 7, 8, 9, 1, 2, 3, 4, 5],
              [7, 8, 9, 1, 2, 3, 4, 5, 6],
              [8, 9, 1, 2, 3, 4, 5, 6, 7],
              [9, 1, 2, 3, 4, 5, 6, 7, 8]])
```

retourne :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que la grille passée en paramètre est valide.
— Aucune des fonctions ne doit modifier la grille passée en paramètre.
— Pour vous aider, nous vous proposons trois exercices intermédiaires, qui vont vous permettre d'écrire et de tester les trois fonctions auxiliaires, avant de les utiliser ici.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.17*.

EXERCICE UPYLAB 6.17a (Vert)

Énoncé

Le sudoku est un jeu de logique dans lequel le joueur reçoit une grille de 9 x 9 cases, chacune pouvant contenir un chiffre de 1 à 9 ou bien être vide. La grille est divisée en 9 lignes, 9 colonnes ainsi qu'en 9 « sous-grilles », appelées régions, formées de 3 x 3 boîtes contiguës. Le but du jeu est de remplir les cases vides avec des chiffres de 1 à 9, de telle sorte que, dans chaque ligne, chaque colonne et chaque région, soient présents tous les chiffres de 1 à 9, sans doublons.

Écrire une fonction `check_rows(grid)` qui prend en paramètre une grille sous forme de matrice à deux dimensions et vérifie si toutes les lignes sont valides (c'est-à-dire que sur chaque ligne, chaque chiffre apparaît une et une seule fois).

Exemple 1

L'appel de la fonction suivant :

```
check_rows([[9, 6, 3, 1, 7, 4, 2, 5, 8],
            [1, 7, 8, 3, 2, 5, 6, 4, 9],
            [2, 5, 4, 6, 8, 9, 7, 3, 1],
            [8, 2, 1, 4, 3, 7, 5, 9, 6],
            [4, 9, 6, 8, 5, 2, 3, 1, 7],
            [7, 3, 5, 9, 6, 1, 8, 2, 4],
            [5, 8, 9, 7, 1, 3, 4, 6, 2],
            [3, 1, 7, 2, 4, 6, 9, 8, 5],
            [6, 4, 2, 5, 9, 8, 1, 7, 3]])
```

retourne :

```
True
```

Exemple 2

L'appel de la fonction suivant :

```
check_rows([[9, 6, 3, 1, 7, 4, 2, 5, 8],
            [1, 7, 8, 3, 2, 5, 6, 4, 9],
            [2, 5, 4, 6, 8, 9, 7, 3, 1],
            [8, 2, 1, 4, 3, 7, 5, 9, 6],
            [4, 9, 6, 8, 4, 2, 3, 1, 7],
            [7, 3, 5, 9, 6, 1, 8, 2, 4],
            [5, 8, 9, 7, 1, 3, 4, 6, 2],
            [3, 1, 7, 2, 4, 6, 9, 8, 5],
            [6, 4, 2, 5, 9, 8, 1, 7, 3]])
```

retourne :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que la grille passée en paramètre est valide.
— La fonction ne doit pas modifier la grille passée en paramètre.

EXERCICE UPYLAB 6.17b (Bleu)

Énoncé

Le sudoku est un jeu de logique dans lequel le joueur reçoit une grille de 9 x 9 cases, chacune pouvant contenir un chiffre de 1 à 9 ou bien être vide. La grille est divisée en 9 lignes, 9 colonnes ainsi qu'en 9 « sous-grilles », appelées régions, formées de 3 x 3 boîtes contiguës. Le but du jeu est de remplir les cases vides avec des chiffres de 1 à 9, de telle sorte que, dans chaque ligne, chaque colonne et chaque région, soient présents tous les chiffres de 1 à 9, sans doublons.

Écrire une fonction `check_cols` qui prend en paramètre une grille sous forme de matrice à deux dimensions et vérifie si toutes les colonnes sont valides (c'est-à-dire que sur chaque colonne, chaque chiffre apparaît une et une seule fois).

Exemple 1

L'appel de la fonction suivant :

```
check_cols([[9, 6, 3, 1, 7, 4, 2, 5, 8],
            [1, 7, 8, 3, 2, 5, 6, 4, 9],
            [2, 5, 4, 6, 8, 9, 7, 3, 1],
            [8, 2, 1, 4, 3, 7, 5, 9, 6],
            [4, 9, 6, 8, 5, 2, 3, 1, 7],
            [7, 3, 5, 9, 6, 1, 8, 2, 4],
            [5, 8, 9, 7, 1, 3, 4, 6, 2],
            [3, 1, 7, 2, 4, 6, 9, 8, 5],
            [6, 4, 2, 5, 9, 8, 1, 7, 3]])
```

retourne :

```
True
```

Exemple 2

L'appel de la fonction suivant :

```
check_cols([[1, 2, 3, 4, 5, 6, 7, 8, 9],
            [2, 3, 4, 5, 6, 7, 8, 9, 1],
            [3, 4, 5, 6, 7, 8, 9, 1, 2],
            [4, 5, 6, 7, 8, 9, 1, 2, 3],
            [5, 6, 7, 8, 9, 1, 2, 3, 4],
            [6, 7, 8, 9, 1, 2, 3, 4, 5],
            [7, 8, 5, 1, 2, 3, 4, 9, 6],
            [8, 9, 1, 2, 3, 4, 5, 6, 7],
            [9, 1, 2, 3, 4, 5, 6, 7, 8]])
```

retourne :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que la grille passée en paramètre est valide.

EXERCICE UPYLAB 6.17c (Bleu)

Énoncé

Énoncé

Le sudoku est un jeu de logique dans lequel le joueur reçoit une grille de 9 x 9 cases, chacune pouvant contenir un chiffre de 1 à 9 ou bien être vide. La grille est divisée en 9 lignes, 9 colonnes ainsi qu'en 9 « sous-grilles », appelées régions, formées de 3 x 3 boîtes contiguës. Le but du jeu est de remplir les cases vides avec des chiffres de 1 à 9, de telle sorte que, dans chaque ligne, chaque colonne et chaque région, soient présents tous les chiffres de 1 à 9, sans doublons.

Écrire une fonction `check_regions` qui prend en paramètre une grille sous forme de matrice à deux dimensions et vérifie si toutes les régions sont valides (c'est-à-dire que dans chaque région, chaque chiffre apparaît une et une seule fois).

Exemple 1

L'appel de la fonction suivant :

```
check_regions([[9, 6, 3, 1, 7, 4, 2, 5, 8],
               [1, 7, 8, 3, 2, 5, 6, 4, 9],
               [2, 5, 4, 6, 8, 9, 7, 3, 1],
               [8, 2, 1, 4, 3, 7, 5, 9, 6],
               [4, 9, 6, 8, 5, 2, 3, 1, 7],
               [7, 3, 5, 9, 6, 1, 8, 2, 4],
               [5, 8, 9, 7, 1, 3, 4, 6, 2],
               [3, 1, 7, 2, 4, 6, 9, 8, 5],
               [6, 4, 2, 5, 9, 8, 1, 7, 3]])
```

retourne :

```
True
```

Exemple 2

L'appel de la fonction suivant :

```
check_regions([[1, 2, 3, 4, 5, 6, 7, 8, 9],
               [2, 3, 4, 5, 6, 7, 8, 9, 1],
               [3, 4, 5, 6, 7, 8, 9, 1, 2],
               [4, 5, 6, 7, 8, 9, 1, 2, 3],
               [5, 6, 7, 8, 9, 1, 2, 3, 4],
               [6, 7, 8, 9, 1, 2, 3, 4, 5],
               [7, 8, 9, 1, 2, 3, 4, 5, 6],
               [8, 9, 1, 2, 3, 4, 5, 6, 7],
               [9, 1, 2, 3, 4, 5, 6, 7, 8]])
```

retourne :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Vous pourrez supposer que la grille passée en paramètre est valide.
— La fonction ne doit pas modifier la grille passée en paramètre.

6.5.9 Exercice UpyLaB 6.18 (Noir)

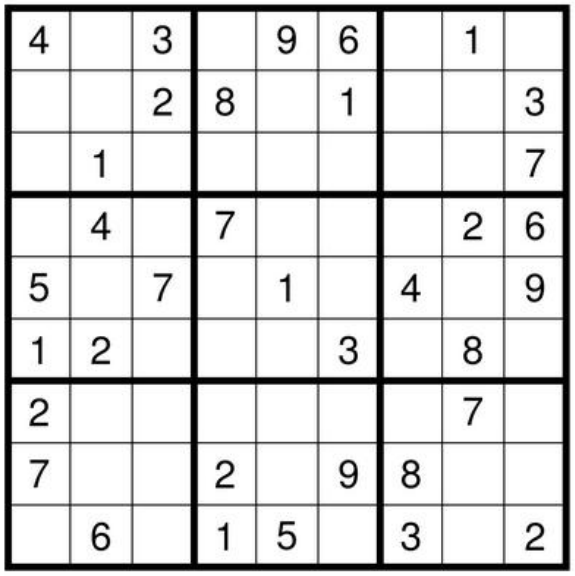
Énoncé

Le sudoku est un jeu de logique dans lequel le joueur reçoit une grille de 9 x 9 cases, chacune pouvant contenir un chiffre de 1 à 9 ou bien être vide. La grille est divisée en 9 lignes, 9 colonnes ainsi qu'en 9 « sous-grilles », appelées régions, formées de 3 x 3 boîtes contiguës. Le but du jeu est de remplir les cases vides avec des chiffres de 1 à 9, de telle sorte que, dans chaque ligne, chaque colonne et chaque région, soient présents tous les chiffres de 1 à 9, sans doublons.

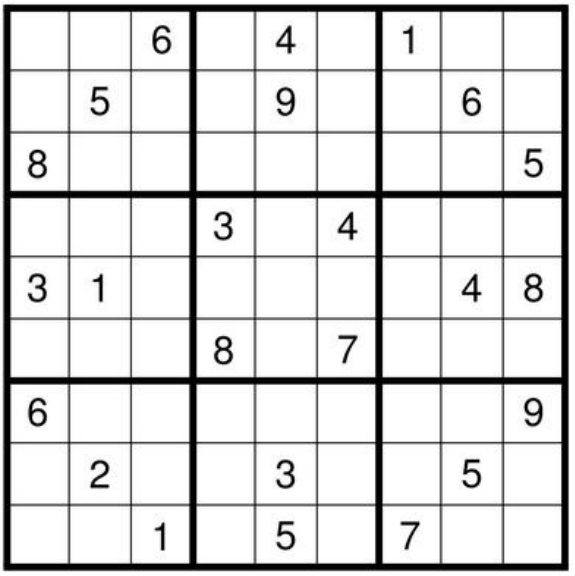
Une méthode de résolution passe par l'analyse des candidats uniques. Ce qu'on appelle candidat est un nombre permis pour une case car on ne le retrouve pas ailleurs dans la ligne, la colonne et la région de cette case.

Si une grille est telle que chaque case vide se retrouve avec un candidat unique, alors la résolution du sudoku est évidente.

Des deux grilles suivantes, seule la première peut être résolue de cette manière : par exemple, en indiçant le tableau à partir de 0, (0,0) étant en haut à gauche, à la case d'indice (4,7), seule la valeur 3 est possible. On peut ensuite faire de même jusqu'à la résolution complète du sudoku.



4		3		9	6		1	
		2	8		1			3
	1							7
	4		7				2	6
5		7		1		4		9
1	2				3		8	
2							7	
7			2		9	8		
	6		1	5		3		2



		6		4		1		
	5			9			6	
8								5
			3		4			
3	1						4	8
			8		7			
6								9
	2			3			5	
		1		5		7		

Écrire une fonction `naked_single(grid)` qui reçoit en paramètre une matrice 9 x 9 d'entiers représentant une grille de sudoku, et qui renvoie un couple de valeurs :

— un booléen `ok` qui indique si la grille peut être résolue en utilisant cette seule méthode du candidat unique,
— la grille complétée si `ok` est égal à `True`, `None` sinon.

Exemple 1

L'appel de la fonction suivant :

```
naked_single([[4, 0, 3, 0, 9, 6, 0, 1, 0],
              [0, 0, 2, 8, 0, 1, 0, 0, 3],
              [0, 1, 0, 0, 0, 0, 0, 0, 7],
              [0, 4, 0, 7, 0, 0, 0, 2, 6],
              [5, 0, 7, 0, 1, 0, 4, 0, 9],
              [1, 2, 0, 0, 0, 3, 0, 8, 0],
              [2, 0, 0, 0, 0, 0, 0, 7, 0],
              [7, 0, 0, 2, 0, 9, 8, 0, 0],
              [0, 6, 0, 1, 5, 0, 3, 0, 2]])
```

retourne :

```
(True, [[4, 7, 3, 5, 9, 6, 2, 1, 8],
        [6, 5, 2, 8, 7, 1, 9, 4, 3],
        [9, 1, 8, 3, 2, 4, 5, 6, 7],
        [3, 4, 9, 7, 8, 5, 1, 2, 6],
        [5, 8, 7, 6, 1, 2, 4, 3, 9],
        [1, 2, 6, 9, 4, 3, 7, 8, 5],
        [2, 9, 5, 4, 3, 8, 6, 7, 1],
        [7, 3, 1, 2, 6, 9, 8, 5, 4],
        [8, 6, 4, 1, 5, 7, 3, 9, 2]])
```

Exemple 2

L'appel de la fonction suivant :

```
naked_single([[0, 0, 6, 0, 4, 0, 1, 0, 0],
              [0, 5, 0, 0, 9, 0, 0, 6, 0],
              [8, 0, 0, 0, 0, 0, 0, 0, 5],
              [0, 0, 0, 3, 0, 4, 0, 0, 0],
              [3, 1, 0, 0, 0, 0, 0, 4, 8],
              [0, 0, 0, 8, 0, 7, 0, 0, 0],
              [6, 0, 0, 0, 0, 0, 0, 0, 9],
              [0, 2, 0, 0, 3, 0, 0, 5, 0],
              [0, 0, 1, 0, 5, 0, 7, 0, 0]])
```

retourne :

```
(False, None)
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Dans la grille passée en paramètre, les cases vides sont représentées par l'entier 0.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

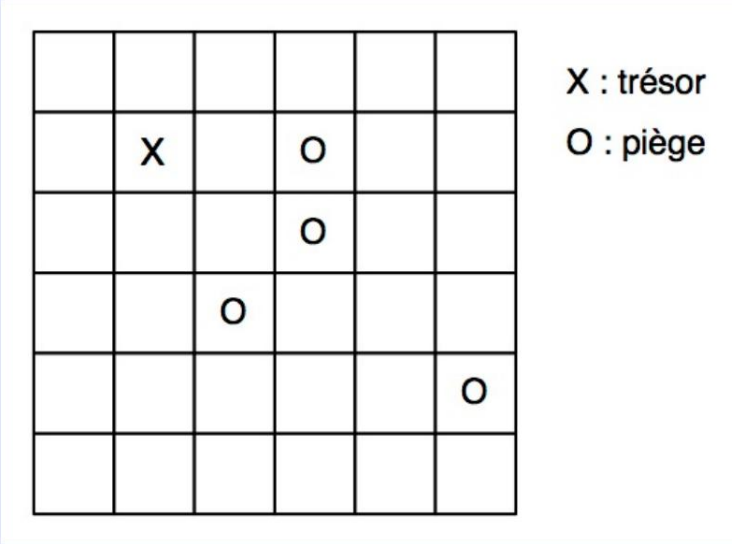
— N'hésitez pas à définir vos propres fonctions annexes.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.18*.

6.5.10 EXERCICE UPYLAB 6.19 (Rouge)

Énoncé

Un dictionnaire peut nous permettre de manipuler des tableaux partiellement remplis, en ne stockant que les cases non vides et en leur associant leur valeur.

Par exemple, le tableau suivant



peut être implémenté par :

```
MY_PRECIOUS = 1
TRAP = -1
my_map = {}
my_map[(2,2)] = MY_PRECIOUS
my_map[(3,4)] = TRAP
my_map[(5,6)] = TRAP
my_map[(2,4)] = TRAP
my_map[(4,3)] = TRAP
```

Notez que les cases vides n'apparaissent pas dans le dictionnaire.

— **Écrire une fonction** `create_map(size, trapsNbr)` qui reçoit deux entiers en paramètres, `size`, compris entre 2 et 100, et `trapsNbr`, de valeur strictement inférieure à `size` x `size`, et qui retourne un dictionnaire implémentant comme dans l'exemple précédent une carte de taille `size` et dans laquelle figurent `trapsNbr` cases contenant un piège (modélisé par la valeur -1) et une case contenant un trésor (modélisé par la valeur 1). L'emplacement de ces cases sera aléatoire.

— **Écrire une fonction** `play_game(map_size, treasure_map)` qui reçoit un entier et une carte de taille `map_size` x `map_size`, telle que celles obtenues grâce à la fonction `create_map`, et qui demande à l'utilisateur d'entrer les coordonnées d'une case, jusqu'à tomber sur une case occupée. Si l'utilisateur a trouvé le trésor, la fonction retourne la valeur `True`, sinon l'utilisateur est tombé sur un piège et la fonction retourne `False`.

Exemple 1

L'appel de la fonction suivant :

```
create_map(4, 5)
```

pourrait retourner :

```
{(3, 1): 1, (4, 2): -1, (1, 1): -1,
 (1, 4): -1, (2, 2): -1, (4, 4): -1}
```

Exemple 2

L'appel de la fonction suivant :

```
play_game(5, {(3, 4): -1, (4, 1): 1, (2, 3): -1, (1, 5): -1})
```

avec les valeurs suivantes saisies en entrée :

```
"4 2 4 4 1 3 4 4 3 1 4 4 4 3 1 1 3 1 3 2 2 1 4 3 1 2 4 1 "
```

retourne :

```
True
```

Exemple 3

L'appel de la fonction suivant :

```
play_game(5, {(3, 4): -1, (4, 1): 1, (2, 3): -1, (1, 5): -1})
```

avec les valeurs suivantes saisies en entrée :

```
"4 7 4 3 a 5 2 3"
```

retourne :

```
False
```

Consignes

— Pour cet exercice, il vous est demandé d'écrire une fonction. Le code que vous soumettrez à UpyLaB ne doit donc comporter que la définition de cette fonction, accompagnée éventuellement de définitions de fonctions annexes, et ne fait en particulier aucun appel ni à la fonction demandée ni aux fonctions `input` et `print`.
— Notez que la numérotation des cases de la carte commence à (1, 1).
— Pour la fonction `create_map`, pensez à utiliser le module random, et le dictionnaire retourné ne comportera que les cases occupées.
— Pour la fonction `play_game`, les coordonnées saisies par l'utilisateur seront données sous la forme d'une série de deux entiers séparés par une espace, une case par ligne.
— Pour la fonction `play_game`, si les valeurs saisies par le joueur ne sont pas du bon type (entières) , ni dans le bon intervalle (comprises entre 1 et `map_size`), ces saisies seront ignorées par le programme.
— Notez enfin que les deux fonctions sont séparées et indépendantes l'une de l'autre.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

— Le dictionnaire généré par la fonction `create_map` ne doit comporter que les cases occupées soit par un piège, soit par le trésor. Les cases vides ne figurent pas dans le dictionnaire.
— Si rien ne marche : consultez la *FAQ sur UpyLaB 6.19*.

6.5.11 EXERCICE UPYLAB 6.20 (Noir)

Énoncé

Arthur nous a envoyé des statistiques produites par UpyLaB contenant des informations sur des exercices et sur des apprenants. Nous souhaitons utiliser ces informations pour classer les exercices.

Chaque jeu de données est formé de deux fichiers de type csv (produit par un tableur du type Excel ou LibreOffice) :

— un fichier `'result-pass-fail.csv'`
— un fichier `'result-count.csv'`

La structure d'un fichier "result-pass-fail-i.csv"

est la suivante :

En première ligne, nous trouvons les intitulés des exercices, chacun séparé du suivant par un caractère point-virgule ";".

Chacune des lignes suivantes correspond aux résultats d'un apprenant : chaque ligne a le même nombre d'éléments que le nombre d'intitulés, et chaque élément vaut soit le texte `'VRAI'`, soit le texte `'FAUX'`, soit est vide (respectivement suivant que l'apprenant a validé tous les tests UpyLaB pour l'exercice correspondant, a demandé la validation par UpyLaB, mais malheureusement ce dernier a répondu par la négative, ou n'a pas encore testé l'exercice dans UpyLaB). À nouveau sur une même ligne, chaque valeur est séparée de la suivante par un caractère point-virgule ";".

La structure du fichier "result-count-i.csv"

est similaire à celle du premier fichier, à la différence près que les textes "VRAI" ou "FAUX" sont remplacés par des nombres naturels strictement positifs donnant le nombre de fois où l'apprenant a testé son code. À nouveau l'entrée est vide si aucune validation n'a été demandée par l'apprenant.

De façon précise, la structure d'un fichier `'result-count-i.csv'` est la suivante :

En première ligne, nous trouvons les intitulés des exercices, chacun séparé du suivant par un caractère point-virgule ";".

Chacune des lignes suivantes correspond aux résultats d'un apprenant : chaque ligne a le même nombre d'éléments que le nombre d'intitulés, et chaque élément représente soit un nombre entier strictement positif, soit est vide (respectivement suivant que l'apprenant a validé ou essayé de valider les tests UpyLaB pour l'exercice correspondant ou n'a pas encore testé l'exercice dans UpyLaB). À nouveau sur une même ligne, chaque valeur est séparée de la suivante par un caractère point-virgule ";".

Écrire un programme qui lit depuis l'entrée deux chaînes de caractères, représentant les noms des deux fichiers décrits ci-dessus (dans l'ordre, le fichier de type "result-pass-fail.csv" suivi du fichier du type "result-count.csv"), et qui imprime la liste des intitulés, un par ligne, dans l'ordre décroissant des « valeurs » calculées comme suit.

La **« valeur »** d'un intitulé est le nombre des « apprenants fiables » ayant réussi l'exercice correspondant.

Un **« apprenant fiable »** est un apprenant qui n'a jamais testé plus de dix fois chacun des codes qu'il a essayé de valider.

Par exemple, si un apprenant n'a testé que trois exercices en soumettant respectivement 1, 2 et 10 essais, il est réputé « fiable ». Si un autre apprenant a testé tous les exercices, mais en soumettant 11 essais pour l'un d'entre eux, il n'est pas considéré comme « fiable ».

Exemple

Avec le fichier `result-pass-fail-0.csv` contenant :

```
ex1;ex2;ex3
VRAI;VRAI;VRAI
VRAI;VRAI;VRAI
VRAI;FAUX;FAUX
VRAI;VRAI;VRAI
VRAI;VRAI;
FAUX;VRAI;VRAI
```

et le fichier `result-count-0.csv` contenant :

```
ex1;ex2;ex3
2;3;5
1;2;4
4;2;666
11;6;3
1;1;
7;7;7
```

le programme affiche :

```
ex2
ex3
ex1
```

En effet, les apprenants 3 et 4 ne sont pas fiables. Ainsi, la valeur de ex2 est égale à 4 ; ex1 et ex3 ont tous les deux une valeur égale à 3 mais ex3 est supérieur à ex1 dans l'ordre utf-8.

Consignes

- Notez que dans cet exercice, il vous est demandé d'écrire un programme qui fera appel aux fonctions `input` et `print`.
- Nous rappelons que votre code sera évalué en fonction de ce qu'il affiche, donc veillez à n'imprimer que le résultat attendu. En particulier, il ne faut rien écrire à l'intérieur des appels à `input` (`input()` et non `input("Entrer un nom de fichier : ")` par exemple), ni ajouter du texte supplémentaire dans ce qui est imprimé (`print(intitules)` et non `print("liste des intitulés :",intitules)` par exemple).
- Les fichiers utilisés par UpyLaB pour tester votre code sont :
  - http://upylab.ulb.ac.be/pub/data/result-pass-fail-0.csv
  - http://upylab.ulb.ac.be/pub/data/result-count-0.csv
  - http://upylab.ulb.ac.be/pub/data/result-pass-fail-1.csv
  - http://upylab.ulb.ac.be/pub/data/result-count-1.csv
- En cas d'égalité, les exercices seront classés selon l'ordre décroissant UTF-8 de leurs intitulés.
- Nous vous conseillons de définir des fonctions annexes que votre programme appellera.
- Le nombre maximal d'essais pour être supposé « apprenant fiable », 10 ici, peut être vu comme une constante globale du programme.

Remarque

Cet exercice vous demande de lire des fichiers csv. Cela peut se faire bien entendu avec ce qui a été vu jusque là dans ce cours, mais vous pouvez aussi consulter la section 6.6 qui présente les outils spécifiques à la manipulation de ces fichiers csv.

AIDE EN CAS DE BESOIN

Vous avez du mal pour réaliser l'exercice. Voici quelques conseils qui peuvent vous aider :

Conseils

- Comme pour les premiers programmes que vous avez écrits lors de ce MOOC, UpyLaB va vérifier que ce qu'imprime votre code correspond exactement à ce qui est attendu. Veillez donc à ne pas ajouter d'espace superflue en fin de ligne, ni de ligne vide supplémentaire. Veillez aussi à ne pas ajouter de texte lors des appels à la fonction `input`.
- Si rien ne marche : consultez la *FAQ sur UpyLaB 6.20*.

6.6 Données persistantes

6.6.1 Présentons le format csv

CSV POUR CRÉER DES DONNÉES PERSISTANTES

Nous avons vu que nous pouvions manipuler des données stockées dans des fichiers texte. Nous pouvons lire des fichiers, récupérer ces données et les traiter. Nous pouvons aussi écrire dans des fichiers pour sauvegarder de façon pérenne les données de nos programmes.

La format `csv` est un format de fichier texte où chaque ligne du fichier contient des données séparées par des virgules. Les lettres CSV signifient *Comma Separated Values*. Ci-dessous un exemple de fichier csv contenant les données d'apprenants d'un MOOC :

```
id,result,attestation
apprenant_1,0.9,1
apprenant_2,0.87,1
apprenant_3,0.43,0
```

Nous constatons que la première ligne du fichier représente les étiquettes des données et que les lignes suivantes stockent les données de chacun des apprenants : son identifiant, son résultat et l'obtention de l'attestation.

Les méthodes et les fonctions vues sur les fichiers, les chaînes de caractères et les dictionnaires nous permettraient de créer, à partir de ces données, les dictionnaires suivants, regroupés dans une liste :

```
[ {'id':'apprenant_1', 'result':0.9, 'attestation':1},
  {'id':'apprenant_2', 'result':0.87, 'attestation':1},
  {'id':'apprenant_3', 'result':0.43, 'attestation':0} ]
```

Le format csv est très répandu et les séparateurs peuvent être la virgule bien sûr mais aussi le point-virgule, l'espace, la tabulation (on parle alors de format *tsv*) ou tout autre caractère choisi par le programmeur et qui **ne doit pas apparaître dans les données**.

Pour aller plus loin, un module python nommé csv existe et facilite la lecture d'un fichier de ce format et la création d'une structure de données adéquate.

CSV CONCRÈTEMENT

Lecture d'un fichier existant

Voici un exemple de fonction qui crée la liste des dictionnaires codant les informations de notre fichier. Cette fonction prend en paramètres un nom de fichier csv et une chaine de caractères représentant le séparateur des données du fichier csv.

```
def charger(filename, sep=','):
    liste_donnees = []
    with open(filename, 'r', encoding='utf-8') as mon_csvfile:
        les_entetes = mon_csvfile.readline().strip().split(sep)
        for ligne in mon_csvfile:
            d_donnees = {}
            l_donnees = ligne.strip().split(sep)
            for index, donnee in enumerate(l_donnees):
                d_donnees[les_entetes[index]] = donnee
            liste_donnees.append(d_donnees)
    return liste_donnees
```

Notez comme la construction de notre liste se fait en deux étapes :

1. La récupération de la première ligne et sa transformation en une liste : ce sont les entêtes des colonnes de nos données, qui serviront aussi à faire les clés de nos dictionnaires
2. la récupération par une boucle, des données des autres lignes et création pour chacune d'elles d'un dictionnaire.

Sur notre exemple introductif, cette liste d'entêtes serait :

```
['id', 'result', 'attestation']
```

Puis, si on considère la deuxième ligne (et donc la première ligne de données), il est important de voir que *"identifiant_1"* correspond à l'entête *"id"*, que *0.9* correspond à *"result"* et que *1* correspond à *"attestation"*. C'est via les indices de nos listes que nous faisons cette correspondance, avec, par exemple l'utilisation de la fonction prédéfinie *enumerate*.

Le module *csv* offre un certain nombre de méthodes pour faciliter les choses. Notons notamment la méthode *DictReader* :

```
def charger(filename):
    with open(filename, 'r', encoding='utf-8') as mon_csvfile:
        reader = csv.DictReader(mon_csvfile)
        return list(reader)
```

Écriture d'un fichier csv

Si nous manipulons des données sous la forme d'une liste de dictionnaires, voici une fonction qui permet de créer un fichier csv stockant ces données :

```
def sauvegarder(filename, l_donnees, sep=','):
    with open(filename, 'w', encoding='utf-8') as file_out:
        l_entetes = [str(e) for e in l_donnees[0]]
        entetes = sep.join(l_entetes)
        file_out.write(entetes + '\n')
        for donnees in l_donnees:
            ligne = sep.join([str(donnees[cle]) for cle in l_entetes])
            file_out.write(ligne + '\n')
```

La construction de la ligne d'entêtes du fichier permet également de récupérer la liste des entêtes, c'est-à-dire des clés des dictionnaires. Chaque dictionnaire correspond alors à une ligne de données dans notre fichier csv. **Attention** lors de l'écriture des données, il faut s'assurer de récupérer les informations dans le **même ordre** que l'ordre des entêtes. C'est pour cela que nous parcourons les clés via la liste *l_entetes*.

Lorsque vos données sont déjà stockées dans un tableur, il est possible d'enregistrer ces données au format csv directement depuis votre application de tableur.

6.6.2 Présentons le module json

JSON POUR CRÉER DES STRUCTURES PERSISTANTES

Une autre possibilité est d'utiliser le format JSON et le module python associé. Le format JSON est également un format texte de fichier qui permet de stocker une structure de données, liste ou dictionnaire. JSON signifie *JavaScript Object Notation*. Il permet l'échange de données entre programmes et un module traitant de ce format est disponible dans la plupart des langages de programmation.

En reprenant l'exemple de nos apprenants, voici un fichier JSON contenant notre liste :

```
[ {"id":"apprenant_1", "result":0.9, "attestation":1},
  {"id":"apprenant_2", "result":0.87, "attestation":1},
  {"id":"apprenant_3", "result":0.43, "attestation":0} ]
```

En Python, le module `json` va nous permettre de lire un fichier pour y récupérer la structure stockée et d'écrire un fichier pour y stocker une structure (liste ou dictionnaire).

LECTURE D'UN FICHIER JSON

Supposons que le fichier texte `mooc.json` contienne les lignes suivantes :

```
[ {"id":"apprenant_1", "result":0.9, "attestation":1},
  {"id":"apprenant_2", "result":0.87, "attestation":1},
  {"id":"apprenant_3", "result":0.43, "attestation":0} ]
```

Dès lors la méthode `load` du module `json` permet de récupérer les données :

```
import json

with open('mooc.json', 'r', encoding='utf-8') as f_in:
    l_apprenants = json.load(f_in)
print(l_apprenants)

[{'id': 'apprenant_1', 'result': 0.9, 'attestation': 1}, {'id': 'apprenant_2', 'result': 0.87,
↪'attestation': 1}, {'id': 'apprenant_3', 'result': 0.43, 'attestation': 0}]
```

Une structure de dictionnaire pourrait être stockée de façon similaire :

```
{ "apprenant_1": {"result":0.9, "attestation":1},
  "apprenant_2": {"result":0.87, "attestation":1},
  "apprenant_3": {"result":0.43, "attestation":0} }
```

Le code Python pour récupérer la donnée serait le même. La différence est que la structure sera un dictionnaire :

```
import json

with open('mooc.json', 'r', encoding='utf-8') as f_in:
    d_apprenants = json.load(f_in)
print(d_apprenants)

{'apprenant_1': {'result': 0.9, 'attestation': 1}, 'apprenant_2': {'result': 0.87, 'attestation
↪': 1}, 'apprenant_3': {'result': 0.43, 'attestation': 0}}
```

ÉCRIRE UN FICHIER JSON

La méthode pour écrire un fichier `json` est `dump`. Voici l'utilisation basique de cette méthode. On suppose qu'un dictionnaire est stocké dans une variable `mon_dict`.

```
import json

with open('donnees.json', 'w', encoding='utf-8') as f_out:
    json.dump(mon_dict, f_out)
```

Dès lors, le fichier texte `donnees.json` sera créé et contiendra une unique ligne :

```
{"apprenant_1": {"result": 0.9, "attestation": 1}, "apprenant_2": {"result": 0.87, "attestation
↪": 1}, "apprenant_3": {"result": 0.43, "attestation": 0}}
```

Un paramètre optionnel nommé `indent` permet de présenter les données dans le fichier avec des indentations, sur plusieurs lignes pour une meilleure lisibilité :

```
import json

with open('donnees.json', 'w', encoding='utf-8') as f_out:
    json.dump(mon_dict, f_out, indent=3)
```

Le fichier `donnees.json` est alors plus lisible :

```
{
    "apprenant_1": {
        "result": 0.9,
        "attestation": 1
    },
    "apprenant_2": {
        "result": 0.87,
        "attestation": 1
    },
    "apprenant_3": {
        "result": 0.43,
        "attestation": 0
    }
}
```

6.7 Bilan du module

6.7.1 Qu'avons-nous vu ?

BREF BILAN DU MODULE

Note : Voir la vidéo de la section 6.4.1 : Rapide bilan du module 6

ERRATUM

Depuis le session 3 de ce MOOC, la présentation du module shelve a été remplacée par celle du module json

BILAN DU MODULE

Dans ce module, nous vous avons présenté deux types de séquences Python : les ensembles (`set`) et les dictionnaires (`dict`).

Un **ensemble** `s` :

- est créé en donnant ses éléments entre accolades `{e1, e2, ...}` ou avec la fonction `set(s)` où `s` est une séquence d'éléments ;
- `set()` crée un ensemble vide ;
- les propriétés mathématiques des ensembles s'appliquent, les doublons sont supprimés ;
- `len(s)` donne le nombre d'éléments de l'ensemble `s` ;
- ayant deux ensembles `a` et `b`, les opérateurs suivants existent
  - `a | b` (union),
  - `a & b` (intersection),
  - `a - b` (différence),
  - `a ^ b` (différence symétrique),

  la différence symétrique prend les éléments dans un des deux ensembles mais pas dans l'autre ;
- la préposition `in` peut également être utilisée comme test d'appartenance ou dans une instruction `for` pour itérer sur tous les éléments de l'ensemble ;
- les opérateurs relationnels (==, !=, <, <=, >, >=) correspondent aux opérateurs d'égalité ou non et d'inclusion stricte ou non dans un sens ou l'autre.

Un dictionnaire `d` :

- est créé en donnant ses éléments *clé : valeur* entre accolade `{k1:v1, k2:v2, ...}` ;
- un dictionnaire vide est créé en mettant deux accolades sans rien à l'intérieur `{}` ;

— chaque clé d'un dictionnaire identifie un élément et peut être de n'importe quel type *non modifiable* (une clé ne peut être une liste ou un ensemble ou un autre dictionnaire). Les valeurs correspondantes peuvent être de n'importe quel type ;
— nous pouvons accéder à une composante de d en écrivant `d[key]` où `key` est une valeur de clé ;
— `d[cle] = valeur` ajoute une composante *cle : valeur* à `d` si elle n'existe pas ou modifie la composante `cle` si elle existe déjà dans d ;
— nous pouvons utiliser la préposition `in` pour tester si une clé fait partie de `d` ou dans un `for` pour itérer sur toutes les clés de `d` ;
— ayant deux dictionnaires `d1` et `d2`, les seuls opérateurs relationnels qui fonctionnent sont l'égalité `d1 == d2` et la différence `d1 != d2` ;
— `del(d[k])` supprime l'élément de `d` de clé `k`.

Nous avons présenté en section 6.2.1 les différentes méthodes permettant de manipuler facilement des dictionnaires :

— `d.clear()` : supprime tous les éléments de `d`
— `d.copy()` : shallow copie de `d`
— `{}.fromkeys(s,v)` : crée un dict avec les clés de `s` et la valeur `v`
— `d.get(k [,v])` : renvoie la valeur `d[k]` si elle existe, `v` sinon
— `d.items()` : liste des items `(k,v)` de `d`
— `d.keys()` : liste des clés
— `d.pop(k [,v])` : enlève `d[k]` s'il existe et renvoie sa valeur ou `v` sinon
— `d.popitem()` : supprime un item `(k,v)` et retourne l'item sous forme de tuple
— `d.setdefault(k [,v])` : `d[k]` si elle existe sinon `v` et rajoute `d[k] = v`
— `d.update(s)` : `s` est une liste de tuples que l'on rajoute à `d`
— `d.values()` : liste des valeurs de `d`

Pour vous aider à les retenir, toutes ces méthodes et leurs fonctionnalités sont reprises dans l" aide-mémoire à votre disposition.

Nous avons aussi vu qu'il est possible de créer un dictionnaire persistant grâce au module `json`.

Nous avons illustré tous ces concepts avec des exemples et la réalisation d'exercices supervisés ou réalisés de façon autonome avec UpyLaB.

6.7.2 Et maintenant ?

TERMINER LES EXERCICES UPYLAB

Il se peut que l'un ou l'autre des exercices UpyLaB que nous vous avons proposés dans nos différents modules de ce cours en ligne ne soit pas terminé. Il se peut aussi que vous ayez maintenant envie d'en faire plus.

En tout les cas, n'oubliez pas l'adage

C'est en forgeant qu'on devient forgeron

mais suivez aussi les conseils de Nicolas Boileau dans L'art poétique :

Avant donc que d'écrire, apprenez à penser.

Ce que l'on conçoit bien s'énonce clairement,

Et les mots pour le dire arrivent aisément.

Hâtez-vous lentement, et sans perdre courage,

Vingt fois sur le métier remettez votre ouvrage,

Polissez-le sans cesse, et le repolissez,

Ajoutez quelquefois, et souvent effacez.

– Nicolas Boileau

Maintenant, si ce n'est déjà fait, terminez tous les exercices UpyLaB que nous vous avons proposés tout au long de ce cours ; ceci afin d'intégrer toute la matière vue dans ce cours en ligne !

Bon travail !

6.8 Bilan global du cours

6.8.1 Qu'avons nous appris dans ce cours ?

QU'AVONS NOUS APPRIS DANS CE COURS ?

Outre le projet je vous voudrez sans doute réaliser, si ce n'est déjà fait, et qui vous permet de faire la synthèse de la plupart des matières vues, nous voici à la fin de ce MOOC. Que de chemin parcouru depuis le début. Nous espérons qu'il vous a plu. Vous y avez réellement appris et acquis une multitude de connaissances et de compétences. Voici ce qui nous vient à l'esprit au premier abord, mais nous sommes persuadés que vous en avez d'autres en tête :

En terme de connaissances

- Python 3 est un langage interprété dont l'apprentissage permet rapidement de réaliser énormément de types de traitements possibles ;
- un code Python manipule des valeurs et variables de différents types (entier, fractionnaire, booléen, séquence de caractères, séquence de valeurs d'autres types, dictionnaire, ensemble,...) ;
- un petit nombre d'instructions différentes permet de réaliser des calculs et traitements divers ;
- des fonctions et méthodes prédéfinies, importées de modules préexistants ou définies dans le programme permettent de structurer le code et de rapidement réaliser les traitements désirés ;
- des fichiers textes peuvent être facilement lus ou créés par un programme Python.

En terme de compétences

- Nous avons également illustré tous ces concepts avec de nombreux exemples et la réalisation d'exercices supervisés ou réalisés avec notre outil UpyLaB ; ceci afin de vous rendre autonome pour résoudre de nouveaux problèmes de traitement de l'information. Vous avez donc acquis une connaissance active de la matière qui vous permet cette autonomie.
- Nous avons également pris le temps de comprendre les mécanismes, par exemple grâce aux diagrammes d'états et l'outil Python Tutor qui permettent de voir exactement ce qui se passe à chaque étape de l'exécution du programme.
- Nous avons aussi donné de nombreux exemples pour montrer une palette de problèmes qui peuvent être traités grâce à la programmation en général et Python en particulier.

6.8.2 Quelles suites possibles à ce cours ?

QUELLES SUITES POSSIBLES À CE COURS ?

Vous avez dès à présent les connaissances pour écrire de nombreux scripts et résoudre des problèmes. Réfléchissez à un problème qui vous tient à coeur et voyez comment le résoudre avec un programme informatique. Lancez-vous ! Vous verrez qu'avec votre bagage actuel vous pourrez déjà envisager de nombreux problèmes.

Parfois vous aurez l'impression qu'il vous manque des outils ou que vous êtes en train de réinventer la roue. Et vous aurez probablement raison. Vous pouvez approfondir vos connaissances en algorithmique ainsi que sur des modules externes existants et offrant une panoplie d'outils supplémentaires optimisés.

Approfondir ses connaissances en algorithmique

Lors de la résolution d'un vrai problème, bien souvent la solution va passer par l'utilisation d'une méthode connue d'où l'idée d'aller explorer ces algorithmes.

De nombreux cours traitent de diverses techniques algorithmiques existentes :

- La récursivité et ses applications
- Les algorithmes de jeux
- La théorie des graphes
- L'apprentissage automatique
- ...

Explorer d'autres modules disponibles en Python

Il existe nombre de petits modules très utiles traitant de choses aussi variées que les arguments sur la ligne de commande, des notions de dates et d'heures, etc.

D'autres modules beaucoup plus gros, issus de la vaste communauté très active méritent qu'on y consacre plusieurs heures. Citons par exemple le module `numpy` pour le calcul numérique, souvent associé à `scipy` pour le calcul scientifique. Ces modules sont complétés par `matplotlib` pour le tracé.

Des modules de parser (analyse de code) vous permettront de compléter efficacement vos connaissances si vous souhaitez traiter des données récupérées automatiquement sur le web.

Voir d'autres paradigmes de programmation

Ce cours vous a initié à la programmation impérative et vous a donné les bases. Vous pouvez aussi appréhender d'autres paradigmes de la programmation : la programmation orientée objet est probablement ce qu'il convient de voir juste après. Python est un langage orienté objet et l'étude des classes est une suite logique à votre premier cours. Vous aurez ainsi l'avantage de rester sur un langage que vous connaissez déjà tout en vous confrontant au monde des objets de façon simple mais néanmoins efficace.

Si vous souhaitez vous orienter dans la programmation de jeux ou de site web dynamique, la programmation événementielle sera incontournable.

La programmation Python ou autre peut toucher encore énormément d'autres domaines !

De toute façon, nous espérons que nous vous avons donné l'envie de continuer dans cette matière ou au moins le désir de comprendre ce qui se passe derrière toutes les applications informatiques que nous utilisons au quotidien dans nos ordinateurs ou nos portables, et pourquoi pas par exemple commencer à comprendre et à créer grâce à vos programmes de l'intelligence artificielle dont on parle tant.

Terminons par deux préceptes importants :

Le premier est de Confucius :

> **Le bonheur ne se trouve pas au sommet de la montagne , mais dans la façon de la gravir.**
>
> – Confucius

Et n'oubliez pas le secret que le renard révéla au petit prince :

> **On ne voit bien qu'avec le coeur. L'essentiel est invisible pour les yeux.**
>
> – Le petit prince - Antoine de Saint-Exupéry

Bien à vous et peut-être à bientôt !

CHAPITRE 7

Votre projet

7.1 Quête dans le château au sommet du Python des Neiges

7.1.1 Information préliminaire

INFORMATION PRÉLIMINAIRE

Le projet que nous vous proposons dans ce MOOC est évalué via une évaluation entre pairs, suivie d'une auto-évaluation. Contrairement aux autres exercices, la réalisation du projet, si vous décidez de la faire, doit donc être synchronisée avec les autres apprenants pour permettre à cette évaluation de se dérouler harmonieusement.

Nous avons décidé de vous proposer 2 sessions pour réaliser votre projet. Une session d'automne (voir calendrier), et une session du printemps. Libre à vous de participer à l'une, l'autre ou même aux deux sessions, sachant que dans ce cas, la note maximale des deux notes qui vous seront attribuées sera retenue dans l'évaluation finale.

Si vous désirez participer à la session d'automne pour le projet, il est temps de réaliser ce dernier.

Ce module donne l'énoncé du projet que nous vous proposons dans le cadre de ce cours en ligne, ainsi que les procédures d'évaluation.

Note : Si vous ne comptez pas faire de projet, nous vous proposons de parcourir quand même cette section ; vous pourrez encore décider de ne pas faire de projet comme vous aviez prévu ou bien d'en faire une partie ou finalement de le faire soit en session d'automne ou plus tard à la session du printemps.

Le projet peut être réalisé dès que vous avez vu la matière du cours jusque et y compris la section 6.1, soit les instructions et fonctions Python, les séquences (chaînes de caractères, tuples et listes) et le début sur les dictionnaires et les ensembles.

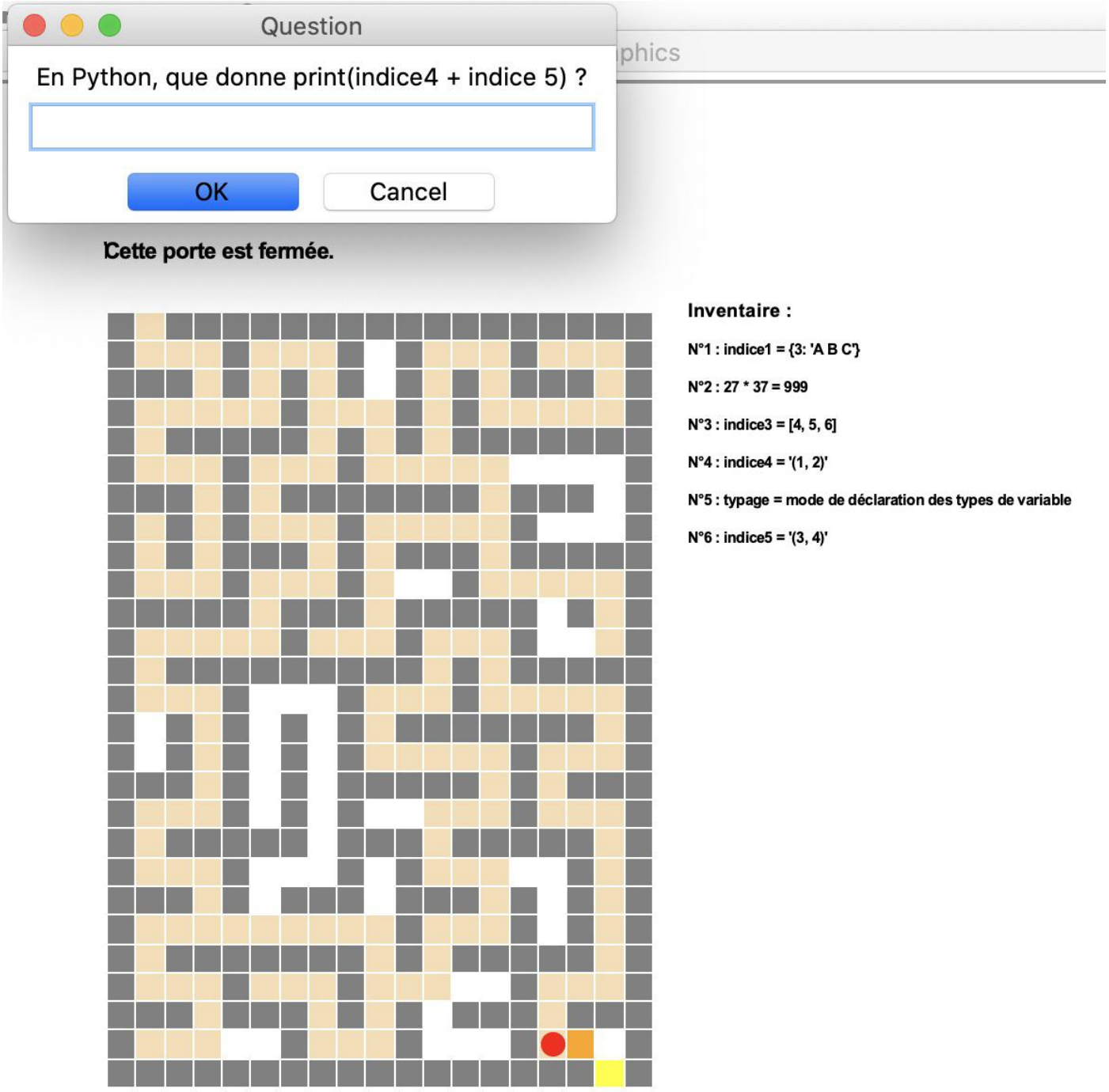
7.1.2 Présentation du projet

Lancelot entre dans le château au sommet du Python des Neiges, muni de son précieux sac de rangement et de sa torche fraîchement allumée aux feux de Beltane. Il doit trouver la statue de sainte Axerror, le chef-d'oeuvre de Gide de Rome, dit le « tyran malfaisant éternel ».

Heureusement, pour l'aider dans sa quête, Merlin, son maître, lui a fourni un plan minutieux des salles et des couloirs du château. Ce plan lui sera fort utile, vu l'énormité du bâtiment, tant par sa taille que par le nombre de ses pièces !

Avant de partir, Merlin lui a donné la clef de la porte d'entrée du château et lui a prodigué moults conseils, dont celui de bien garder tous les objets qu'il trouvera lors de sa quête : ceux-ci lui permettront de répondre aux diverses énigmes que ne manqueront pas de poser les gardes postés devant les portes à l'intérieur du château.

Merlin a affirmé à son disciple que, s'il procède avec intelligence, sa quête sera satisfaite.


Question
En Python, que donne print(indice4 + indice 5) ?
OK
Cancel
Cette porte est fermée.
Inventaire :
N°1 : indice1 = {3: 'A B C'}
N°2 : 27 * 37 = 999
N°3 : indice3 = [4, 5, 6]
N°4 : indice4 = '(1, 2)'
N°5 : typage = mode de déclaration des types de variable
N°6 : indice5 = '(3, 4)'

Le jeu en une image

EN BREF

Ce projet, si vous le menez jusqu'au bout, va vous faire programmer un petit jeu du type *jeu d'évasion* (*escape game*) dans lequel le joueur commande au clavier les déplacements d'un personnage au sein d'un « château » représenté en plan. Le château est constitué de cases vides (pièces, couloirs), de murs, de portes, que le personnage ne pourra franchir qu'en répondant à des questions, d'objets à ramasser, qui l'aideront à trouver les réponses à ces questions et de la case de sortie / quête du château. Le but du jeu est d'atteindre cette dernière.

Le programme utilisera le module turtle comme interface graphique et comportera deux parties principales :

1. le tracé du plan du château,
2. la gestion du jeu sur le plan tracé (gestion des déplacements du personnage au sein du château, affichage des objets recueillis, gestion de l'ouverture des portes).

Les données nécessaires (plan du château, objets, portes) sont encodées dans trois fichiers texte. Vous disposerez d'un jeu de fichiers de données que nous vous proposons et devrez réaliser un programme de jeu qui met en œuvre ces données. Vous pourrez ensuite si vous le souhaitez encoder votre propre château en préparant d'autres fichiers de données, qui tourneront avec le même programme. Votre château pourra aussi bien être du type labyrinthe (entièrement fait de couloirs étroits, sans portes) que du type escape game (un ensemble de pièces, entre lesquelles un personnage circule pour rassembler des objets lui permettant de répondre à des questions ou résoudre des énigmes).

Ce projet vous permettra d'écrire un code plus substantiel que ce que nous vous demandons ailleurs dans le cours, et va vous demander de manipuler de nombreux concepts vus tout au long du cours.

Il vous est demandé que votre programme puisse gérer divers jeux de données construits sur les mêmes principes que les nôtres. Cela permettra à chacun des participants de tester son programme sur les données proposées par d'autres.

NIVEAUX DE PROJET

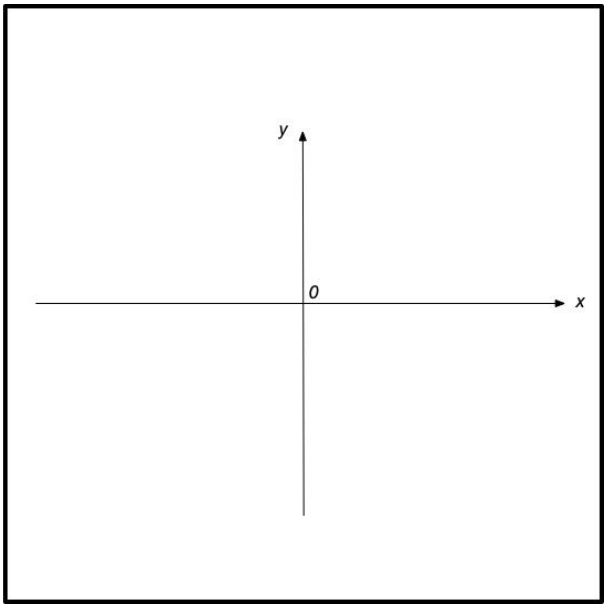
Dans le cadre de ce projet, 4 niveaux de jeux peuvent être réalisés correspondant à des difficultés croissantes :

- **Niveau 1 : construction et affichage du plan du château** Ce niveau consiste à tracer correctement le plan du château à partir du fichier de données. La gestion du jeu après affichage du plan ne fait pas partie de ce niveau.
- **Niveau 2 : en plus du niveau 1, la gestion des déplacements au clavier, sans portes ni objets à ramasser** Ce niveau consiste à gérer les déplacements d'un personnage dans le plan construit au niveau 1. Ici on suppose qu'aucun objet ni porte n'est présent dans le plan : leur gestion n'est pas prise en compte.
  - On déplace le personnage de case en case à l'aide des 4 flèches du clavier.
  - Le personnage ne doit pas pouvoir traverser les murs ni sortir du plan.
  - Si le personnage arrive sur la case quête / sortie du château, un message lui annonce qu'il a gagné.
  - **Option non demandée, si vous voulez aller plus loin :** les cases déjà parcourues par le personnage peuvent être affichées dans une couleur spécifique, de façon à ce que la trace de son parcours soit conservée.
- **Niveau 3 : niveaux 1 + 2 + gestion des objets à ramasser** Dans ce niveau, il y a dans le labyrinthe des objets que le joueur doit collecter. Outre le fichier contenant le plan, un second fichier de données contient la liste des objets et la case où chaque objet se trouve.
  - Les cases où se trouve un objet doivent apparaître dans une couleur spécifique.
  - Lorsque le joueur se déplace sur une case contenant un objet, un message lui signale qu'il a trouvé un objet, et ce dernier s'ajoute à l'inventaire affiché en permanence à côté du plan. La case où était l'objet devient vide, puisque l'objet a été ramassé.
  - **Option non demandée, si vous voulez aller plus loin :** on peut prévoir que la sortie du labyrinthe ne sera accessible qu'une fois tous les objets rassemblés.
- **Niveau 4 (escape game complet) : niveaux 1 + 2 + 3 + gestion des portes** Ce niveau consiste à ajouter dans la labyrinthe des portes que le joueur doit ouvrir en répondant à des questions. Le troisième fichier de données contient la liste des portes avec les questions et réponses associées.
  - Les cases où se trouve une porte doivent apparaître dans une couleur spécifique.
  - Lorsque le joueur tente d'accéder à une porte, la question associée lui est posée.
  - S'il répond correctement, la porte s'ouvre et il peut alors la franchir. La case de la porte apparaît alors comme une case vide.
  - S'il ne répond pas ou s'il donne une mauvaise réponse, la porte reste fermée et infranchissable.

7.1.3 En pratique

SYSTÈME DE COORDONNÉES

Le module turtle utilise un système de coordonnées dont l'origine `(0, 0)` est au centre le la fenêtre. L'axe des abscisses est orienté vers la droite, l'axe des ordonnées est orienté vers le haut.



Coordonnées des points dans une fenêtre turtle

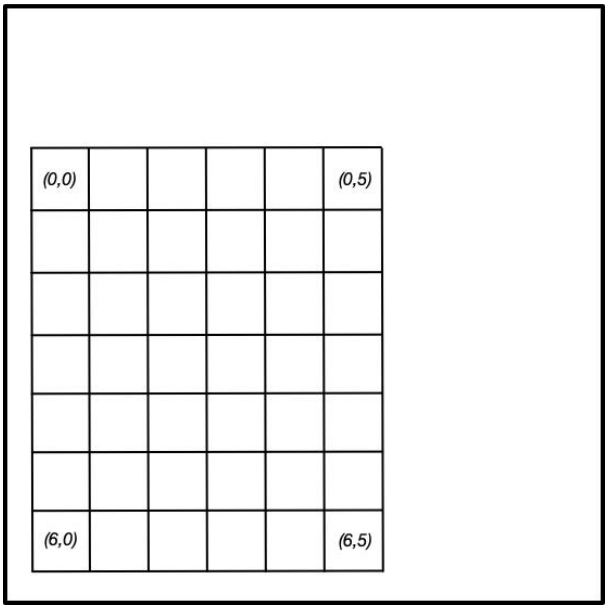
Dans ce qui suit, nous parlerons de *pixel turtle* pour parler de coordonnées dans la fenêtre turtle. Nous utiliserons les *pixels turtle* de `(-240, -240)` à `(240, 240)`.

Pour faciliter les discussions, nous vous demandons de numéroter les cases du plan du château :

— en numérotant les lignes de haut en bas et les colonnes de gauche à droite,

— en commençant la numérotation des lignes comme des colonnes à partir de 0 (convention habituelle de python).

La case (0, 0) est donc située en haut et à gauche du plan et la case `(0, 5)` est située 5 cases à sa droite. la case `(6, 5)` sera la 6e case depuis la gauche, sur la 7e ligne depuis le haut.



Coordonnées des carrés du plan du château

Point important : Lorsque vous calculerez les coordonnées nécessaires pour tracer une case, souvenez-vous que le numéro de ligne augmente quand l'ordonnée « y » (pour la fenêtre turtle) diminue.

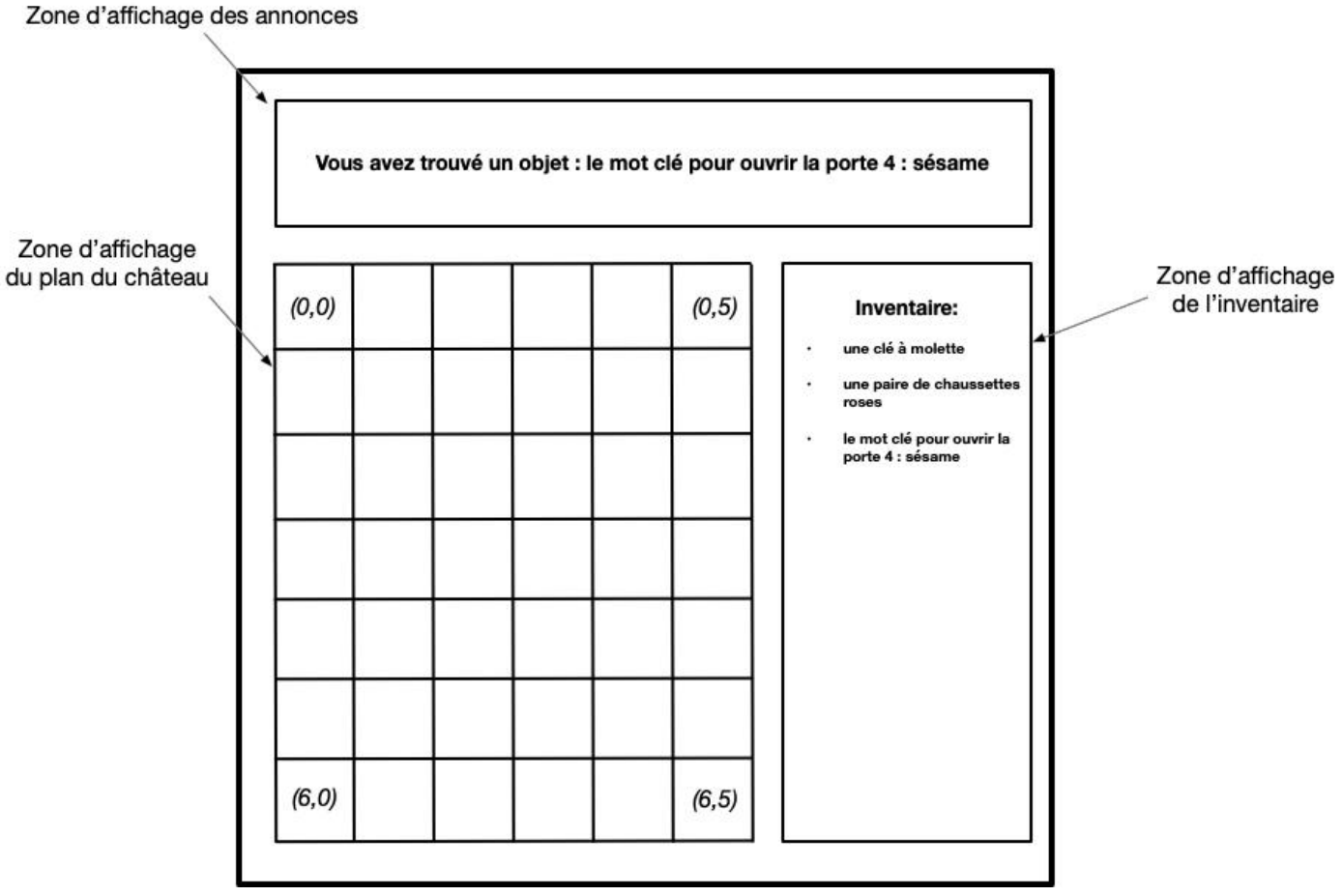
OCCUPATION DES ZONES DE LA FENÊTRE TURTLE

Vous aurez besoin de distinguer trois zones différentes dans la fenêtre turtle :

— un bandeau en haut pour l'affichage d'annonces, qui peut par exemple faire 40 *pixels turtle* de hauteur, que vous pourrez définir par les coordonnées en *pixels turtle* du début de la ligne de texte,

— en-dessous, une large zone à gauche pour l'affichage du plan, qu'il sera pratique de définir par les coordonnées de son coin inférieur gauche (abscisse et ordonnée minimales de la zone) et de son coin supérieur droit (abscisse et ordonnée maximales de la zone),

— la partie restant à droite sera la colonne d'affichage de l'inventaire, que vous pourrez définir par les coordonnées en *pixels turtle* du début de la première ligne de texte de l'inventaire.


Zone d'affichage des annonces
Vous avez trouvé un objet : le mot clé pour ouvrir la porte 4 : sésame
Zone d'affichage du plan du château
(0,0)
(0,5)
(6,0)
(6,5)
Inventaire:
une clé à molette
une paire de chaussettes roses
le mot clé pour ouvrir la porte 4 : sésame
Zone d'affichage de l'inventaire

Zones d'affichage du jeu dans la fenêtre turtle

DESCRIPTION DES FICHIERS DE DONNÉES

Les données correspondant à un jeu figurent dans trois fichiers texte dont la structure est décrite ci-dessous.

a. Un fichier texte `plan_chateau.txt` qui contient les données du plan du château.
   Il comporte un certain nombre de lignes, toutes de la même longueur, représentant une ligne de cases du plan. Chaque ligne du fichier est une suite d'entiers séparés par des espaces, où chaque entier représente une case. Les valeurs de ces entiers codent chaque nature de case :
   — valeur 0 pour une case vide,
   — valeur 1 pour un mur (infranchissable),
   — valeur 2 pour la case de sortie/victoire,
   — valeur 3 pour une porte qui sera franchissable en répondant à une question,
   — valeur 4 pour une case contenant un objet à collecter.

b. Un fichier texte `objets.txt` qui contient une liste d'objets associés aux cases sur lesquelles on les trouve.
   Chaque ligne sera du type :
   `(x, y) , "objet"`
   Ainsi, chaque ligne contient :
   — un couple d'entiers positifs ou nuls (numéro de ligne, numéro de colonne) indiquant la case où se trouve l'objet,
   — puis une virgule et une espace,
   — puis une chaînes de caractères décrivant l'objet.
   Par exemple :
   `(12, 3), "un oreiller magique"`
   signifiera que la case de coordonnées `(12, 3)` contient l'objet « un oreiller magique ».

c. Un fichier texte `portes.txt` qui contient une liste de questions/réponses associées aux portes.
   Chaque ligne sera du type :
   `(x, y) , ("question", "réponse")`
   Ainsi, chaque ligne contient :
   — un couple d'entiers positifs ou nuls (numéro de ligne, numéro de colonne) indiquant la case où se trouve la porte,
   — puis une virgule et une espace,
   — puis un couple de chaînes de caractères avec une question et une réponse.
   Par exemple :
   `(21, 12), ("Capitale de la Belgique ?", "Bruxelles")`
   signifiera que pour franchir la porte située en case `(21, 12)`, il faudra répondre « Bruxelles » à la question « Capitale de la Belgique ? ».

CONSTANTES ET JEU DE DONNÉES TYPE

Pour vous aider à concevoir et à tester votre programme, un jeu de données comportant trois fichiers types à téléchartger sur le format spécifié plus haut est donné ici :
— fichier `plan_chateau.txt` (https://upylab.ulb.ac.be/pub/Projets_MOOC/plan_chateau.txt)
— fichier `dico_objets.txt` (https://upylab.ulb.ac.be/pub/Projets_MOOC/dico_objets.txt)
— fichier `dico_portes.txt` (https://upylab.ulb.ac.be/pub/Projets_MOOC/dico_portes.txt)

Il s'agit d'un escape game très simple avec des questions portant sur Python. Nous vous suggérons de placer ces trois fichiers dans le même dossier que votre programme, pour pouvoir les appeler sans devoir décrire leur place dans l'arborescence.

Nous supposons aussi que la porte d'entrée du château est ouverte (elle est donc vue comme une case vide), qu'elle est en position `(0, 1)` et que le personnage s'y trouve initialement.

Un fichier `CONFIGS.py` ((https://upylab.ulb.ac.be/pub/Projets_MOOC/CONFIGS.py) vous est fourni : il donne les dimensions, couleurs et quelques constantes utilisées par notre programme de référence. Vous pouvez télécharger ce fichier et copier/coller ces définitions dans votre script ou mieux placer ce fichier `CONFIGS.py` dans le même répertoire que votre script et ajouter à ce dernier la ligne

```
from CONFIGS import *
```

dont l'effet en pratique sera le même qu'un copier/coller.

```
ZONE_PLAN_MINI = (-240, -240)  # Coin inférieur gauche de la zone d'affichage du plan
ZONE_PLAN_MAXI = (50, 200)  # Coin supérieur droit de la zone d'affichage du plan
POINT_AFFICHAGE_ANNONCES = (-240, 240)  # Point d'origine de l'affichage des annonces
POINT_AFFICHAGE_INVENTAIRE = (70, 210)  # Point d'origine de l'affichage de l'inventaire

# Les valeurs ci-dessous définissent les couleurs des cases du plan
COULEUR_CASES = 'white'
COULEUR_COULOIR = 'white'
COULEUR_MUR = 'grey'
COULEUR_OBJECTIF = 'yellow'
COULEUR_PORTE = 'orange'
COULEUR_OBJET = 'green'
COULEUR_VUE = 'wheat'
COULEURS = [COULEUR_COULOIR, COULEUR_MUR, COULEUR_OBJECTIF, COULEUR_PORTE, \
            COULEUR_OBJET, COULEUR_VUE]
COULEUR_EXTERIEUR = 'white'

# Couleur et dimension du personnage
COULEUR_PERSONNAGE = 'red'
RATIO_PERSONNAGE = 0.9  # Rapport entre diamètre du personnage et dimension des cases
POSITION_DEPART = (0, 1)  # Porte d'entrée du château

# Désignation des fichiers de données à utiliser
fichier_plan = 'plan_chateau.txt'
fichier_questions = 'dico_portes.txt'
fichier_objets = 'dico_objets.txt'
```

Nous supposons donc que la fenêtre turtle est contenue dans un rectangle (de coin inférieur gauche en `(-240, -240)` et de coin supérieur droit `(240, 240)`)) et est composée de trois zones :

— la zone *affichage du plan* (de coin inférieur gauche en `(-240, -240)` et de coin supérieur droit `(50, 200)`),
— la zone *annonces* avec les textes qui seront affichés en `(-240, 240)` (chaque annonce devra effacer la précédente),
— la zone *affichage de l'inventaire* (de coin **supérieur gauche** en `(70, 210)` et de coin **supérieur droit** en `(240, 210)`)

Pour faire simple, nous supposons également que le personnage démarre à la case de coordonnée (1,0)

7.2 Détails des phases du projet

Nous sommes conscients que ce projet, si vous voulez le faire dans sa totalité, est ambitieux surtout pour des programmeurs débutants.

Nous vous proposons donc, pour mener à bien votre projet, de le phaser dans sa réalisation ; de créer dans un premier temps (niveau 1) un embryon de projet, qui va grossir petit à petit (niveaux 2, 3) pour finalement englober la totalité de ce qui vous est demandé.

7.2.1 Niveau 1 : construction et affichage du plan du château

Le niveau 1 parle de lecture du plan à partir du fichier `plan_chateau.txt`, de la construction de la matrice correspondante pour stocker ce plan, et de son affichage grâce au module `turtle`.

Pour cela différentes fonctions doivent être programmées.

FONCTION DE LECTURE DU FICHIER CONTENANT LE PLAN

Vous devez écrire une première fonction `lire_matrice(fichier)`, qui recevra en argument le nom d'un fichier texte contenant le plan à tracer. Elle ouvrira ce fichier et renverra en sortie une matrice, c'est-à-dire une liste de listes, soit une liste dont chaque élément sera lui-même une liste représentant une ligne horizontale de cases du plan.

Prenons l'exemple de ce plan comprenant 4 lignes et 6 colonnes avec les cases vides blanches, des murs gris, un objet orange et une porte verte :

Plan d'un chateau très simple

Il est représenté par le fichier suivant, composé d'entiers séparés par des espaces et de retours à la ligne :

```
1 0 1 1 1 1
1 0 0 0 0 1
1 0 4 0 0 1
1 1 1 3 1 1
```

La fonction de lecture du fichier renverra la liste de listes suivante :

```
[[1, 0, 1, 1, 1, 1], [1, 0, 0, 0, 0, 1], [1, 0, 4, 0, 0, 1], [1, 1, 1, 3, 1, 1]]
```

FONCTIONS D'AFFICHAGE DU PLAN

Pour tracer le plan du château à partir de la matrice obtenue, vous écrirez une fonction `afficher_plan(matrice)` utilisant le module `turtle` ; cette fonction recevra en entrée la matrice telle que décrite ci-dessus.

Pour réaliser cette fonction, nous vous conseillons de suivre les étapes suivantes :

1. Commencer par une fonction `calculer_pas(matrice)` qui calcule la dimension à donner aux cases pour que le plan tienne dans la zone de la fenêtre `turtle` que vous avez définie : diviser la largeur et la hauteur de la zone en question par le nombre de cases qu'elle doit accueillir, retenir la plus faible de ces deux valeurs.

   Par exemple, pour une zone turtle d'affichage pour le plan de 290 (-240 à 50) de large et 440 (-240 à 200) de haut, si le plan fait 20 cases en largeur et 44 cases en hauteur, les carrés auront une taille de 10 pour ne pas sortir de la zone (ici le souci, si la taille est supérieure à 10, sera la hauteur du plan).
2. Définir une fonction `coordonnees(case, pas)` qui calcule les coordonnées en *pixels turtle* du coin inférieur gauche d'une case définie par ses coordonnées (numéros de ligne et de colonne). Souvenez-vous que les lignes sont numérotées de haut en bas, alors que l'axe des coordonnées verticales va de bas en haut. Par exemple : avec un château de 44 lignes (lignes 0 à 43), la case inférieure gauche du château (c'est-à-dire la case `(43, 0)`) pour une sous-fenêtre turtle allant de `(-240, -240)` à `(50, 200)` vaudra `(-240, -240)`.
3. Définir une fonction `tracer_carre(dimension)`, traçant un carré dont la dimension en *pixels turtle* est donnée en argument.
4. Définir une fonction `tracer_case(case, couleur, pas)`, recevant en arguments un couple de coordonnées en indice dans la matrice contenant le plan, une couleur, et un pas (taille d'un côté) et qui va appeler la fonction `tracer_carre` pour tracer un carré d'une certaine couleur et taille à un certain endroit.

5. Définir enfin une fonction `afficher_plan(matrice)`, qui va appeler la fonction `tracer_case` pour chaque ligne et chaque colonne du plan, par deux boucles imbriquées. Le principe est : pour chaque élément ligne de la matrice, pour chaque élément colonne de cet élément ligne, tracer une case à l'emplacement correspondant, dans une couleur correspondant à ce que dit la matrice.

CORPS DU PROGRAMME AU NIVEAU 1

Il restera alors à écrire la suite d'instructions qui va :
- fixer les valeurs nécessaires (noms des fichiers de données, couleurs des divers types de cases, points définissant les différentes zones de l'écran),
- appeler les différentes fonctions nécessaires pour créer la matrice et tracer le plan,

et à réaliser un premier script complet, contenant l'ensemble des éléments (docstrings, import, définitions des constantes, des fonctions et corps du programme) tel qu'expliqué dans le manuel des bonnes pratiques

7.2.2 Niveau 2 : gestion des déplacements

Bravo si vous avez terminé ce niveau 1. Vous pouvez passer au niveau 2. Ici, nous vous demandons d'ajouter à votre programme la gestion du déplacement d'un personnage.

Le personnage sera représenté par un petit rond (fonction prédéfinie turtle.dot). Il pourra se déplacer case par case dans les 4 directions (haut, bas, droite, gauche).
- Lorsque le personnage tentera un déplacement vers une case de mur ou vers l'extérieur du plan, rien ne se produira.
- Lorsqu'il tentera un déplacement vers une case vide, il avancera d'une case (en pratique, il s'agira de retracer la case de départ pour effacer le personnage, et de replacer ce dernier sur la case de destination).

Pour réaliser cette partie du code nous vous suggérons les parties suivantes :

FONCTION DE DÉPLACEMENT DU PERSONNAGE

Nous vous suggérons de définir une fonction principale de gestion des déplacements, qui sera assez simple pour le niveau 2, et que vous viendrez compléter si vous passez aux niveaux 3 (gestion des objets) et 4 (gestion des portes).

Pour le niveau 2, donc, nous avons besoin d'une fonction `deplacer(matrice, position, mouvement)` qui reçoit en arguments : - `matrice` : le plan du château, - `position` : un couple définissant la position où se trouve le personnage, - `mouvement` : un couple définissant le mouvement demandé par le joueur.

Pour l'instant (niveau 2) cette fonction aura les effets suivants :
- Si le mouvement souhaité fait sortir le personnage des limites du plan, rien ne se passera.
- Si le mouvement souhaité mène le personnage sur un mur, rien ne se passera.
- Si le plan que vous utilisez comprend des objets ou des portes, vous traiterez les objets comme des cases vides et les portes comme des cases de mur.

Note : Au niveau 2, le traitement des portes et des objets trouvés ne sont pas traités, et sont « vus » comme des murs et des cases vides. Ces éléments seront traités correctement aux niveaux 3 et 4, expliqués plus bas.

SAISIE DES DÉPLACEMENTS

Pour gérer la commande du personnage au clavier, nous allons utiliser de la programmation événementielle. Le principe est d'associer à un événement un certain traitement. Ici, il faudra associer au fait de pousser sur une touche du clavier (les flèches de votre clavier) un traitement qui gère le déplacement associé de votre personnage.

Pour ceci, nous allons utiliser des fonctions `turtle` qui n'ont pas été vues dans le MOOC, et nous vous donnons donc ci-dessous les portions de code nécessaires :

Pour faire avancer le personnage grâce aux touches du clavier

1. Nous allons utiliser :

   — la fonction `turtle.listen()` pour demander à Python « d'écouter » ce qui se passe sur le clavier,
   — la fonction `turtle.onkeypress()` pour associer une touche du clavier au déclenchement d'une fonction (notons que cette fonction ne peut avoir de paramètres),
   — la fonction `turtle.mainloop()` pour placer le programme en position d'attente d'une action.

   Vous placerez donc, dans la partie traitement (corps) de votre programme, le bloc suivant, qui associe aux 4 flèches du clavier (qui sont identifiées dans `turtle` avec les noms (chaînes de caractères) `"Left"` (flèche vers la gauche), `"Right"` (flèche vers la droite), `"Up"` (flèche vers le haut) et `"Down"` (flèche vers le bas)) les 4 fonctions respectives `deplacer_gauche`, `deplacer_droite`, `deplacer_haut` et `deplacer_bas`. Nous verrons au point suivant que, dans notre programme, il faudra définir chacune de ces 4 fonctions pour qu'elle déclenche le traitement requis chaque fois que la touche clavier correspondante est enfoncée. Ainsi, lorsqu'une des touches de flèche sera pressée (« onkeypress »), la fonction associée sera appelée.

```
turtle.listen()    # Déclenche l'écoute du clavier
turtle.onkeypress(deplacer_gauche, "Left")    # Associe à la touche Left une fonction␣
↪appelée deplacer_gauche
turtle.onkeypress(deplacer_droite, "Right")
turtle.onkeypress(deplacer_haut, "Up")
turtle.onkeypress(deplacer_bas, "Down")
turtle.mainloop()    # Place le programme en position d'attente d'une action du␣
↪joueur
```

2. Vous devrez alors définir les 4 fonctions (`deplacer_gauche()`, `deplacer_droite()`, `deplacer_haut()`, `deplacer_bas()`) que déclenchent les 4 flèches du clavier. Mais si le joueur appuie de manière répétée et rapide sur une touche, ou en continu en gardant le doigt enfoncé, la fonction associée risque d'être lancée plusieurs fois en parallèle, ce qui poserait de grandes difficultés. Aussi, nous allons encadrer le bloc d'instructions de la fonction par des instructions visant à désactiver provisoirement la touche concernée. Pour cela, nous allons associer provisoirement la touche à la valeur `None` qui représente une absence de valeur (et donc de traitement associé).

   Vous placerez donc dans votre code une définition de fonction du type qui suit, en ce qui concerne la flèche gauche, et des fonctions similaires pour les 3 autres flèches :

```
def deplacer_gauche():
    turtle.onkeypress(None, "Left")    # Désactive la touche Left
    ... # traitement associé à la flèche gauche appuyée par le joueur
    turtle.onkeypress(deplacer_gauche, "Left")    # Réassocie la touche Left à la␣
↪fonction deplacer_gauche
```

3. Il reste un point auquel il faut veiller sur le fonctionnement de la fonction `turtle.listen()`.

   Nous aurons besoin, pour le niveau 4, que le programme pose une question au joueur lorsque celui-ci veut franchir une porte. Pour cela, nous vous proposerons d'utiliser la fonction `turtle.textinput()` qui affiche une fenêtre de saisie puis attend que le joueur y entre une chaîne de caractères. Cette fenêtre de saisie lance en fait son propre `turtle.listen()` associé à la fenêtre de saisie, ce qui interrompt le `turtle.listen()` que nous avons lancé auparavant. Il faut donc, après l'utilisation de `turtle.textinput()`, relancer le `turtle.listen()`.

   En pratique, retenez ceci : si vous utilisez la fonction `turtle.textinput()`, vous devrez placer une nouvelle ligne `turtle.listen()` juste après, pour que le programme continue à détecter l'appui sur les touches du clavier.

```
        reponse = turtle.textinput("Question", dico_portes[case][0])
        turtle.listen()
```

7.2.3 Niveau 3 : collecte d'objets dans le labyrinthe

Pour le niveau 3, nous vous demandons d'ajouter au programme tel que rédigé au niveau 2 la gestion des objets, présents dans le château, que le personnage collecte quand il passe sur la case correspondante. Pour cela les éléments supplémentaires suivants devront être codés.

FONCTION DE LECTURE DES FICHIERS CONTENANT LES OBJETS

Vous aurez besoin d'une fonction `creer_dictionnaire_des_objets(fichier_des_objets)` créant un dictionnaire d'objets à partir du fichier correspondant, présenté plus haut. Cette fonction recevra en argument le nom du `fichier_des_objets`, et renverra un dictionnaire comportant :

— en clefs les couples `(ligne, colonne)` désignant les cases où se trouvent les objets,
— en valeurs les chaînes de caractères correspondant aux objets.

Ainsi le fichier d'objets suivant :

```
(12, 3), "un oreiller"
(3, 15), "une paire de ciseaux"
```

donnera le dictionnaire :

```
{(12,3): "un oreiller", (3, 15): "une paire de ciseaux"}
```

Note :

La lecture des données reçues du fichier des objets (et on verra plus loin, aussi le fichiers des portes) peut être facilitée par l'utilisation de verbe `eval`. Par exemple, ayant une chaîne de caractères `chaine = '(1, 2), "bonjour"'`, après l'instruction :

```
a, b = eval(chaine)
```

`a` vaudra `(1, 2)` et `b` vaudra `"bonjour"`.

COMPLÉMENT À LA FONCTION DE DÉPLACEMENT DU PERSONNAGE

Au niveau 2, vous avez défini une fonction gérant le déplacement, qui examine si le personnage est envoyé vers une case vide ou vers un mur. Vous devez maintenant ajouter un autre cas à cette fonction, celui où le personnage est envoyé vers une case contenant un objet, et donc définir une fonction `ramasser_objet` :

— L'objet disparaîtra de la case (à la fois dans le plan et à l'affichage), qui devra donc prendre la couleur des cases vides.
— Le personnage avancera sur la case demandée.
— Une annonce du type « Vous avez trouvé : une clef à molette » s'affichera dans le bandeau d'affichage des annonces.
— L'objet s'ajoutera à l'inventaire des objets collectés affiché dans la colonne d'affichage de l'inventaire.

Pour cela, vous aurez sans doute besoin de passer de nouveaux paramètres à la fonction gérant le déplacement : la matrice contenant le plan (puisque vous serez amené à modifier la nature de la case contenant l'objet lorsque celui-ci sera ramassé) et l'ensemble contenant l'inventaire des objets ramassés (puisque l'objet sera ajouté à cet ensemble).

7.2.4 Niveau 4 : Le jeu escape game complet avec questions-réponses

Le niveau 4 demande de réaliser le jeu complet tel qu'annoncé en début d'énoncé. Pour cela les éléments supplémentaires suivants devront être codés.

FONCTION DE LECTURE DES FICHIERS CONTENANT LES QUESTIONS/RÉPONSES

Vous aurez besoin de lire le fichier contenant les questions-réponses associées aux portes. Cela peut être réalisé par la fonction de lecture du fichier des objets (`creer_dictionnaire_des_objets`) que vous aurez écrite au niveau précédent, puisque la structure des fichiers est presque la même.

Ainsi le fichier de questions/réponses suivant :

```
(12, 3) ("3 + 2 = ?", "5")
(3, 15) ("Quel était le prénom d'Henri IV ?", "Henri")
```

donnera le dictionnaire :

```
{(12,3): ("3 + 2 = ?", "5"),
 (3, 15): ("Quel était le prénom d'Henri IV ?", "Henri")}
```

COMPLÉMENT À LA FONCTION DE DÉPLACEMENT DU PERSONNAGE

Vous allez maintenant devoir ajouter le cas où la case de destination souhaitée est une porte. Vous aurez besoin d'une fonction `poser_question(matrice, case, mouvement)` pour :

- afficher dans le bandeau d'annonces `Cette porte est fermée.`
- poser au joueur la question correspondant à l'emplacement de la porte et saisir sa réponse,
- si la réponse est bonne, remplacer la porte par une case vide, afficher dans le bandeau d'annonce que la porte s'ouvre, et avancer le personnage,
- si la réponse est mauvaise, l'annoncer et ne pas déplacer le personnage.

Pour poser une question, vous pouvez utiliser la fonction `turtle.textinput()`. Elle prend en argument 2 chaînes de caractères, une qui sera le titre de la fenêtre d'input (par exemple « Question ») et une autre qui sera le texte affiché dans la fenêtre d'input (la question associée à la porte).

Comme cela a été signalé plus haut, utiliser la fonction `turtle.textinput()` interrompt l'écoute du clavier déclenchée par `turtle.listen()`, et vous devrez placer sur la ligne suivante une nouvelle instruction `turtle.listen()` pour recommencer à surveiller le clavier.

7.2.5 Remise et évaluation du projet

CONSIGNES À RESPECTER POUR LE PROJET

Consignes sur le contenu du projet

Les consignes décrites dans les sous-sections précédentes sont à respecter scrupuleusement ; relisez-les attentivement avant la remise !

Echéances

Consultez le calendrier pour connaître les dates limites

- de remise de votre projet ainsi que
- celle du travail d'évaluation des projets de vos pairs.

Attention

Si vous avez choisi de réaliser votre projet pour la session d'automne, vous devez respecter les échéances de cette session (soumission, évaluation des projets des pairs, seconde soumission, auto-évaluation). Dans le cas contraire, la session de printemps sera votre dernière possibilité, de nouveau en respectant les différentes échéances.

Notez que si vous n'avez pas soumis à temps votre projet pour la première phase d'évaluation par les pairs, il vous sera quand même possible de soumettre votre travail pour la phase suivante d'auto-évaluation, mais alors vous vous noterez sur 24 points et non sur les 48 points possibles pour le projet.

Modalités de remise

La procédure de remise est expliquée à la section suivante.

MAIS COMMENT ÉVALUER UN PROJET ?

De façon binaire nous pourrions dire qu'un programme informatique est bon s'il fait ce qu'on lui demande et sinon il est incorrect. Ce serait un peu court surtout après avoir parlé de toutes les règles de bonnes pratiques que nous avons vues.

En fait, un projet tel que le projet Château est évalué selon plusieurs critères de correction (programme correct, qui fait bien son travail) et de style (code qui suit les règles de bonnes pratiques).

Donnons ici des recommandations pour évaluer un projet de programmation niveau débutant.

Principes de base

La réalisation d'un projet informatique comprend plusieurs aspects qui font l'objet de choix qui doivent généralement être justifiés, ou de mise en application, généralement sous forme de programme structuré.

Tout d'abord, il est important que l'évaluation soit le résultat d'une notation
- positive : qui consiste à donner des points quand certains aspects sont présents dans la solution, et
- négative : qui consiste à enlever des points sur certains aspects qui sont absents ou incorrects.

Il convient aussi in fine d'avoir une évaluation globale où un projet qui *fonctionne* se voit *naturellement* attribuer des points même si plusieurs problèmes existent dans la réalisation.

Le mieux pour obtenir une telle note qui est le fruit d'une évaluation **négative et positive**, est de découper la note finale en sous notes, qui chacune évalue un aspect, qui de même fait l'objet d'une évaluation *négative et positive* avec évaluation globale.

Proposition

Ci-dessous la découpe de la note à donner au projet, que nous vous demandons d'appliquer dans vos évaluations pour obtenir une notation uniforme pour tous. Le nombre de points pour chaque catégorie est proposée pour une note finale sur 24 points. Comme il s'agit d'un projet pour débutant, les structures de données et les méthodes pour résoudre les problèmes sont données et ne nécessitent pas de recherche pour l'apprenant ; aucune note n'est donc donnée pour ces deux aspects.

Grille de notation du projet (sur 24 points)

- **Découpe (4 points)** [La découpe du code global avec commentaires] initiaux, et respect de l'ordre des parties : import des modules externes, suivi des définitions des constantes globales, suivi des définitions des fonctions et enfin, code global.
  - 4 points si les quatre parties sont présentes et l'ordre est respecté ;
  - 3 points si une des quatre parties manque ou est dans le mauvais ordre ;
  - 2 points si deux des quatre parties manquent ou sont dans le mauvais ordre ;
  - 1 points si trois des quatre parties manquent ou sont dans le mauvais ordre ;
  - 0 point si rien n'est satisfaisant.
- **Structuration en fonctions (3 points)** [La structuration en] fonctions de bonne taille (25 lignes maximum) et cohérentes (pas de fonctions définies mais non appelées par exemple) est réalisée :
  - 3 points si la structuration est satisfaisante ;
  - 2 points si des fonctions sont définies, mais certaines sont incohérentes soit sont trop longues ;
  - 1 points si des fonctions sont définies, mais certaines sont incohérentes et d'autres sont trop longues ;
  - 0 point si aucune fonction n'est définie ou une certaine structuration en fonction existe mais ne respecte pas du tout les règles de bonnes pratiques.
- **Bonnes pratiques (4 points)** [Les bonnes pratiques sont] respectées en matière d'utilisation des noms des constantes, variables, fonctions…, de l'indentation… :
  - 4 points si toutes les bonnes pratiques sont respectées ;
  - 3 points si les noms des constantes, variables et fonctions respectent les bonnes pratiques, mais l'indentation (4 caractères par incrément) n'est pas respectée ;
  - 2 points si les règles sont globalement repectées à quelques exceptions près à la fois au niveau indentations et noms de constantes, variables ou fonctions ;

    - 0 point si les règles de bonnes pratiques en matière d'encodage ne sont globalement pas respectées.
- **Commentaires (4 points)** [Commentaires et docstrings sont bien] présents : le docstring initial possède les informations suivantes : identité de l'auteur, date, ce que fait le programme, les fonctions possèdent un docstring décrivant les paramètres, ce que fait la fonction aux paramètres (par défaut, ils ne sont pas modifiés) et le résultat de la fonction, des commentaires pertinents existent quand c'est utile à la compréhension
    - 4 points si commentaires et docstrings sont présent et respectent les règles de bonnes pratiques ;
    - 3 points si un des éléments du docstring initial ou des commentaires est manquant ou non satisfaisant ;
    - 2 points si le docstring initial et les commentaires sont manquants ou non satisfaisants, mais que les fonctions ont des docstrings corrects ;
    - 0 point si les trois éléments docstring initial, commentaires et docstrings de fonctions sont manquants ou non satisfaisants.
- **Consignes (2 points)** : (voir énoncé du projet)
    - 2 points si toutes les consignes du projet sont respectées (niveau 4 de l'énoncé atteint) ;
    - 1 point si une bonne partie des consignes est respectée (niveau 2 ou 3 atteint) ;
    - 0 point si peu de consignes sont respectées (le niveau 1 n'a pas été dépassé).
- **Résultat (7 points)** : Le programme fonctionne correctement.
    - 7 points si le projet complet fonctionne toujours correctement (niveau 4 de l'énoncé) ;
    - 6 points si le programme donne les bons résultats à quelques exceptions près (niveau 3 atteint) ;
    - 4 points si certaines parties du projet ne sont pas (bien) réalisées (niveau 2 atteint) ;
    - 2 points si une partie du projet seulement est réalisée (niveau 1 atteint) ;
    - 0 point si le programme ne s'exécute pas sans échec ou si aucune partie ne fonctionne.

Explication sur l'évaluation

En plus des points attribués, il est essentiel pour l'évaluateur d'expliquer pourquoi telle ou telle note a été donnée, pour que l'auteur du projet puisse comprendre les notes données et puisse s'améliorer les fois suivantes.

ET COMMENT ALLEZ-VOUS ÊTRE ÉVALUÉ SUR VOTRE PROJET ?

Dans le cadre de ce cours, comme annoncé, l'évaluation de votre projet, se fera en deux temps :

- d'une part par une évaluation par les pairs
- d'autre part, par une auto-évaluation de votre projet après éventuelles améliorations.

En pratique, la remise du projet et son évaluation seront faites en **6 étapes** :

1. Après l'avoir réalisé, vous soumettrez votre projet.
2. Vous devrez évaluer trois projets d'autres étudiants choisis au hasard.
3. Vous recevrez les notes et commentaires sur votre propre projet venant d'au moins deux autres de vos pairs (deux autres apprenants).
4. À la lumière des commentaires donnés par vos pairs, vous corrigerez et soumettrez une version améliorée de votre projet.
5. Vous réaliserez une auto-évaluation, c'est-à-dire évaluerez votre propre projet.

Le document consignes pour la notation, téléchargeable ici reprend les explications et détaille chaque notation et demande de commenter chaque note donnée.

La note finale de votre travail sera la somme des évaluations par les pairs (24 points) et de votre auto-évaluation (24 points).

Les détails pratiques sur les évaluations seront donnés plus loin.

Temps et délais pour réaliser le projet

L'estimation du temps moyen que va vous mettre le projet Château est très variable. La matière vue jusqu'ici suffit pour sa réalisation. Vous avez plusieurs jours pour sa remise. Libre à vous de mener en parallèle la suite du module 6 avec la réalisation du projet ou plutôt de postposer la suite de l'apprentissage du module 6 et de vous atteler dès maintenant au projet.

DÉTAILS PRATIQUES SUR L'ÉVALUATION

Les paragraphes qui suivent vous permettent de réaliser :
- d'une part l'évaluation des projets de trois de vos pairs
- et d'autre part, quand vous aurez reçu deux évaluations de votre propre projet (d'autres pairs), une auto-évaluation de votre projet.

DERNIERS CONSEILS AVANT DE FAIRE LES ÉVALUATIONS

Vous et tous les autres apprenants débutez en programmation. L'évaluation par les pairs avec les commentaires qui l'accompagnent ainsi que l'auto-évaluation peuvent être très utiles pour votre apprentissage.

Pour que cela soit le cas, lors des différentes phases de notation, essayez d'être :
- objectif dans vos évaluations,
- clair et précis dans vos commentaires,
- mais toujours courtois et bienveillant

comme vous voudriez que les autres le soient pour vous-même.

Bon travail !

7.3 Phase d'évaluation par les pairs

7.3.1 Phase d'évaluation par les pairs

Si vous comptez réaliser votre projet pour la session d'automne, cette section et la section suivante contient les procédures pour soumettre votre, projet, réaliser et rapporter vos évaluations sur les projets des pairs que nous vous demandons d'effectuer soumettre à nouveau votre projet et finalement l'auto-évaluer.

MENU DE CETTE SOUS-SECTION

Cette sous-section contient trois parties qui vont vous demander des actions :

1) D'abord vous allez soumettre votre projet éventuellement accompagné d'un lien vers l'image d'un résultat de votre programme.
2) Ensuite vous allez évaluer le projet de trois de vos pairs (ou plus si demandé et si vous le voulez bien) et remettre des notes et vos commentaires les plus précis possible.
3) Enfin dans cette partie, vous allez recevoir la note des évaluations d'au moins deux de vos pairs avec les détails des notes et les commentaires associés.

24 points seront en jeu dans cette partie.

La suite (resoumission de votre projet éventuellement amendé et auto-évaluation) est l'objet de la sous-section qui suivra.

Note : Voir le formulaire d'évaluation du projet par les pairs dans le cours en ligne

---

7.4 Phase d'auto-évaluation du projet

7.4.1 Phase d'auto-évaluation

MENU DE CETTE SOUS-SECTION

Cette sous-section contient trois parties qui vont vous demander des actions :

1) D'abord vous allez soumettre une nouvelle fois votre projet éventuellement amendé suite à la lecture des notes et commentaires de vos pairs à nouveau éventuellement accompagné d'un lien vers l'image d'un résultat de votre programme.
2) Ensuite vous allez évaluer votre propre projet que vous venez de soumettre et remettre des notes et vos commentaires sur ce dernier.
3) Enfin, vous allez recevoir la note de votre auto-évaluation.

À nouveau, 24 points seront en jeu dans cette partie, ce qui totalisera 48 points pour la réalisation de votre projet.

Note : Voir le formulaire d'auto-évaluation du projet dans le cours en ligne

---

CHAPITRE 8

Projet donné lors des trois premières sessions du MOOC Apprendre à coder avec Python

8.1 Le projet Vasarely

8.1.1 Énoncé du projet

LE PROJET : INFORMATION PRÉLIMINAIRE

Nous donnons ici l'énoncé du projet que nous avons proposé dans le cadre du cours en ligne (FUN-MOOC) « Apprendre à coder avec Python », ainsi que les procédures d'évaluation proposées aux apprenants dans le cadre d'une évaluation par les pairs.

Le projet couvre les instructions et fonctions Python ainsi que le BA-ba des séquences (chaînes de caractères et tuples) ce qui correspond à la matière du MOOC jusque et y compris la section 5.1.

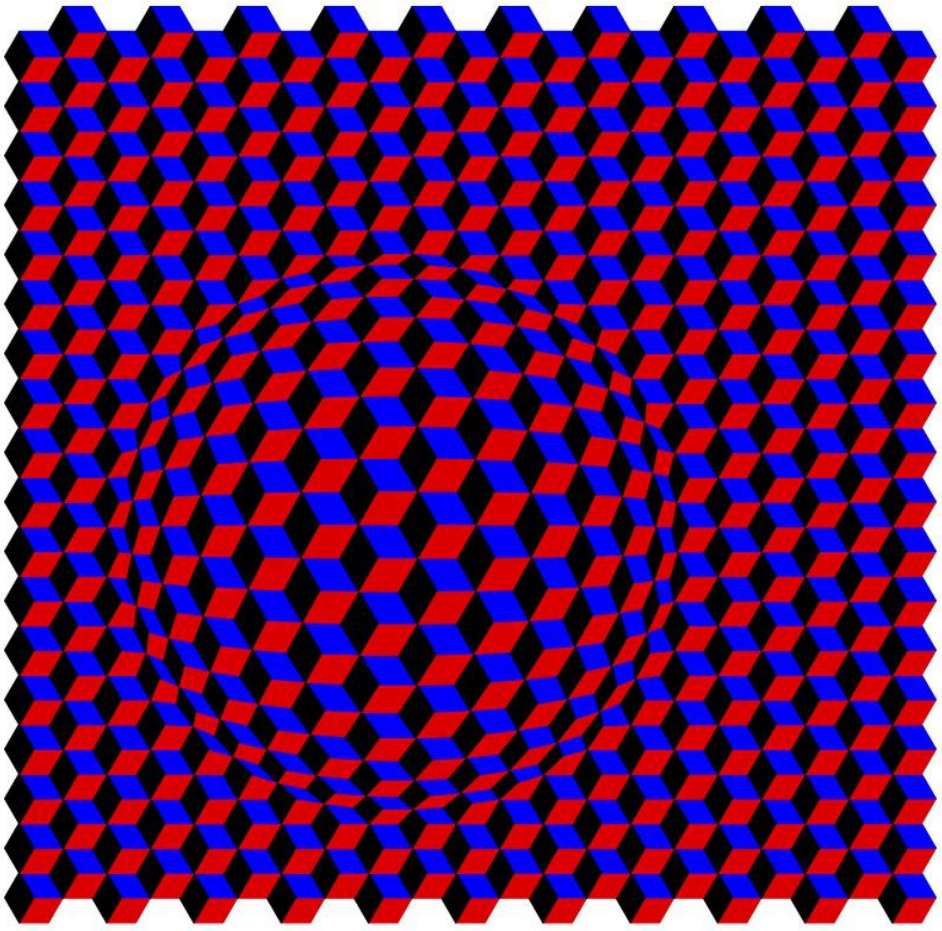
PROJET VASARELY (OP ART)

Dans ce qui suit, nous donnons l'énoncé de ce projet, baptisé « projet Vasarely » en référence à l'artiste Victor Vasarely qui a popularisé cette tendance. (Pour en savoir plus sur cet artiste, n'hésitez pas à consulter la page dédiée suivante : Victor Vasarely).

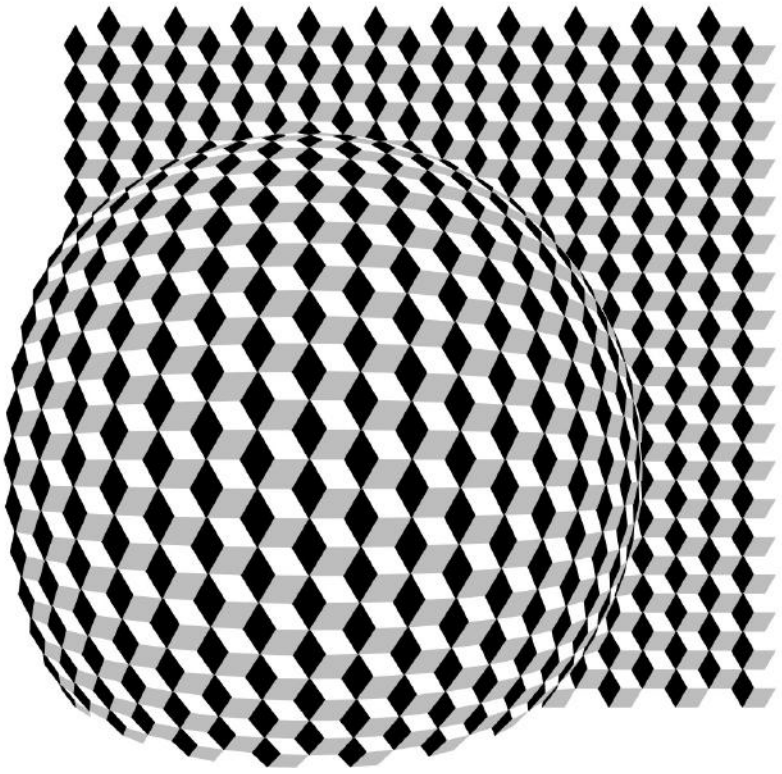
ÉNONCÉ DU PROJET VASARELY

Brève description du projet

Il vous est demandé de réaliser un programme en Python produisant des tableaux d'art optique comme représentés par les figures ci-dessous avec différentes couleurs et déformations :



Exemple de pavage déformé (bleu, rouge, noir)



Autre exemple de pavage déformé (noir, blanc, gris)

Ces tableaux représentent des pavages hexagonaux, vus d'en haut, formés avec des losanges de couleurs différentes, déformés par une boule. Votre programme utilisera le module `turtle`, présenté au module 2 de ce cours, pour peindre vos tableaux.

8.1.2 En pratique

LES CONSIGNES POUR VOTRE PROJET

Nous travaillons avec des coordonnées et distances telles que manipulées par les différentes fonctions de `turtle` (`up`, `begin_fill`, `goto`, ...). En particulier nous demandons de tracer, avec turtle, la vue de dessus d'une figure en 3 dimensions et donc de manipuler tantôt des coordonnées en 3 dimensions sous forme de triplet (x, y, z), tantôt en 2 dimensions sous forme de couple (x, y).

Nous vous demandons de réaliser quatre étapes.

ÉTAPE 1

En supposant que turtle est déjà importé, nous vous demandons d'écrire une fonction

```
hexagone(point, longueur, col, centre, rayon)
```

qui reçoit :

- le point `point` sous forme d'un triple (tuple de trois composantes) donnant la valeur des trois coordonnées du centre avant déformation de l'hexagone à peindre,
- la distance (avant déformation) `longueur` entre le centre et n'importe quel sommet de l'hexagone,
- le tuple `col` contenant les trois couleurs `(col1, col2, col3)` qui vont être utilisées pour dessiner les hexagones,
- le point `centre` sous forme de triple `(c_x, c_y, c_z)` qui donne le centre de la sphère de déformation,
- et le `rayon` de la sphère de déformation.

La fonction peint un hexagone déformé en traçant des lignes droites entre le centre et les extrémités dont la position est calculée avec la fonction `deformation`.

`col1` représente la couleur du pavé Nord-Est (en haut à droite), `col2` représente la couleur du pavé Ouest (à gauche) et `col3` représente la couleur du pavé Sud-Est (en bas à droite).

Note : Pour tester votre fonction, vous pouvez dans un premier temps définir la fonction `deformation` comme suit :

```
def deformation(point, centre, rayon):
    """renvoie le point sans le modifier"""
    return point
```

qui renvoie le point `point` sans le modifier.

ÉTAPE 2

Écrire ou télécharger la fonction `deformation(point, centre, rayon)` avec :
- `point` (`tuple` à 3 composantes `(p_x, p_y, p_z)`) : le point à déformer
- `centre` (`tuple (c_x, c_y, c_z)`) : le point central de la sphère de déformation
- `rayon` (`float`) : le rayon de la sphère de déformation

et sachant qu'elle renvoie le point `(x2, y2, z2)` après déformation.

Si vous décidez d'utiliser notre version de la fonction deformation, vous pouvez la copier / coller dans votre code à l'endroit adéquat (voir règles de bonnes pratiques).

```
from math import pi, sin, cos, sqrt, acos, asin, atan2

def deformation(p, centre, rayon):
    """ Calcul des coordonnées d'un point suite à la déformation engendrée par la sphère␣
↪émergeante
        Entrées :
          p : coordonnées (x, y, z) du point du dalage à tracer (z = 0) AVANT déformation
          centre : coordonnées (X0, Y0, Z0) du centre de la sphère
          rayon : rayon de la sphère
        Sorties : coordonnées (xprim, yprim, zprim) du point du dallage à tracer APRÈS␣
↪déformation
    """
    x, y, z = p
    xprim, yprim, zprim = x, y, z
    xc, yc, zc = centre
    if rayon**2 > zc**2:
        zc = zc if zc <= 0 else -zc
        r = sqrt(
            (x - xc) ** 2 + (y - yc) ** 2)                    # distance horizontale depuis le␣
↪point à dessiner jusqu'à l'axe de la sphère
        rayon_emerge = sqrt(rayon ** 2 - zc ** 2)             # rayon de la partie émergée de la␣
↪sphère
        rprim = rayon * sin(acos(-zc / rayon) * r / rayon_emerge)
        if 0 < r <= rayon_emerge:                          # calcul de la déformation dans les autres cas
            xprim = xc + (x - xc) * rprim / r                 # les nouvelles coordonnées sont␣
↪proportionnelles aux anciennes
            yprim = yc + (y - yc) * rprim / r
        if r <= rayon_emerge:
            beta = asin(rprim / rayon)
            zprim = zc + rayon * cos(beta)
            if centre[2] > 0:
                zprim = -zprim
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return (xprim, yprim, zprim)

if __name__ == "__main__": # code de test
    for i in range(-150,150,50):
        for j in range(-150,150,50):
            print(deformation((i,j,0), (0,0,100), 100))
        print()
```

Ou vous pouvez importer le fichier `module_deformation.py` en cliquant sur le lien ci-dessous pour ensuite ouvrir le fichier `module_deformation.py` et copier / coller la définition de la fonction `deformation` dans votre code du projet Vasarely.

Note : Une alternative au copier / coller serait de faire un import de la fonction.

Pour ceux qui veulent essayer de définir `deformation` eux-mêmes (**attention : la théorie et le code sont beaucoup plus difficiles** que tout ce que nous avons fait jusqu'à présent) :

voici les explications de base en cliquant ici

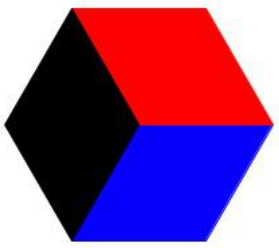
et voici quelques conseils ou remarques :

— Vous pouvez utiliser la librairie `math`.
— Les calculs se font sur des points en trois dimensions.
— Le pavage avant déformation se trouve sur le plan avec z = 0.

Note : Notons que nous nous simplifions le problème lors du tracé des hexagones déformés : en effet après calculs des déplacements des points extrémités grâce à la fonction `deformation`, le programme tracera des lignes droites (et non pas des lignes déformées) entre ces extrémités. Ne vous en faites pas : l'effet final n'en est que peu affecté.

Conseil : pour tracer les hexagones utilisez la méthode `goto(x,y)` de `turtle` plutôt que `forward`.

Exemple d'hexagone avant et après transformation



Exemple de pavé hexagonal non déformé

hexagone de centre (0, 0, 0) de longueur d'arête avant déformation 30 avec déformation de centre (-50, -50, 0) et de rayon 90 (en rose, les contours du pavé avant déformation)

ÉTAPE 3

Écrire une fonction :

`pavage(inf_gauche, sup_droit, longueur, col, centre, rayon)`

qui reçoit :

- des valeurs entières `inf_gauche`, `sup_droit` précisant les coordonnées d'une fenêtre dont le coin inférieur gauche est `(inf_gauche, inf_gauche)` et le coin supérieur droit est `(sup_droit, sup_droit)` sachant que l'on ne prend en compte que les axes x et y, la hauteur du pavage avant transformation étant égale à 0 ;
- la `longueur` entre le centre et n'importe quel sommet de chaque hexagone avant déformation ;
- les trois couleurs sous forme de triple `col = (col1, col2, col3)` ;
- le `centre` de la déformation utile pour la fonction `deformation` (`centre = (x_c, y_c, z_c)` ;
- le `rayon` de la sphère de déformation utile pour la fonction `deformation`.

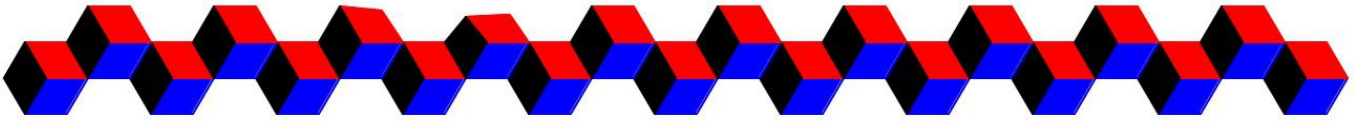
La fonction `pavage` peint les hexagones déformés dont les centres, **avant déformation**, se trouvent à l'intérieur de la fenêtre (bords inclus) avec l'hexagone en bas à gauche, avant déformation, centré sur le point `(inf_gauche, inf_gauche)`. Pour cela, elle utilise la fonction `hexagone`.

La séquence de couleurs est la même pour tous les pavés du pavage (tous les losanges en haut à droite du pavé sont de `col1`, à gauche, de `col2` et en bas à droite, de `col3`).

Que vous définissiez vous-même la fonction `deformation` ou que vous preniez la nôtre, de toute façon, n'oubliez pas que `turtle` ne tient compte que des deux premières dimensions, la valeur de z des points n'étant pas envoyée par exemple aux `goto(x,y)`.

Les figures suivantes montrent comment votre fonction pavage peut agencer les dessins des pavés, ligne par ligne.

Première ligne inférieure



Deux premières lignes inférieures peintes

Les couleurs possibles sont données par la spécification de couleur Tk, voir : Couleurs et Couleurs2.

ÉTAPE 4

Enfin, écrire un code principal qui demande à l'utilisateur les paramètres de l'exécution et qui appelle la fonction `pavage` pour réaliser le résultat.

Les paramètres seront donnés comme suit :

- `inf_gauche` : (valeur entière) donnant les coordonnées `(inf_gauche, inf_gauche)` du bord inférieur gauche de la fenêtre de visualisation ;
- `sup_droit` : (valeur entière) donnant les coordonnées `(sup_droit, sup_droit)` du bord supérieur droit de la fenêtre de visualisation ;
- `longueur` : (valeur entière) longueur d'un segment de pavé (avant déformation) ;
- `col` : (trois chaîne de caractères) donnant les trois couleurs des pavés ;

— `centre` : (trois entiers) donnant les coordonnées du centre de la sphère déformante ;
— `r` : (entier) donnant le rayon de la sphère déformante.

Par exemple les entrées de votre programme seront :

```
-305
305
20
blue
black
red
-50
-50
-50
200
```

Contrairement aux exercices UpyLaB, ici vos instructions *ìnput*` peuvent afficher le texte qu'elles désirent ; par exemple une exécution pourrait donner l'affichage sur la console suivant :

```
Coin inférieur gauche (val, val) : -305
Coin supérieur drout  (val, val) : 305
Longueur d'une arrête : 20
Couleur 1 : blue
Couleur 2 : black
Couleur 3 : red
Abscisse du centre du cercle : -50
Ordonnée du centre du cercle : -50
Hauteur du centre du cercle : -50
Rayon du cercle : 200
```

Pour vous permettre de montrer l'image produite par votre programme, après la réalisation du travail, soit vous capturez l'image résultat, soit votre programme demandera à `turtle` de sauver le résultat dans un fichier. Pour cela, votre programme pourra exécuter en fin de programme les deux instructions :

```
turtle.getcanvas().postscript(file="pavage.eps")
turtle.done()
```

où ici le nom du fichier est `"pavage.eps"`. Notez que le seul format de fichier possible pour turtle est postscript (suffixe eps). Si, après l'exécution du programme, vous désirez un autre format pour le fichier résultat (jpeg ou pdf par exemple) il vous suffira de faire traduire le résultat, par exemple en utilisant une traduction en ligne (avec votre navigateur web, cherchez **online translation eps to jpg** par exemple).

Au terme de votre projet, si vous le désirez, un exemple de résultat de votre code, avec les couleurs et déformations que vous désirez, pourra être publié et visible par tous. Utilisez le **forum du Projet** pour toute question concernant l'énoncé.

EXEMPLES DE RÉSULTATS

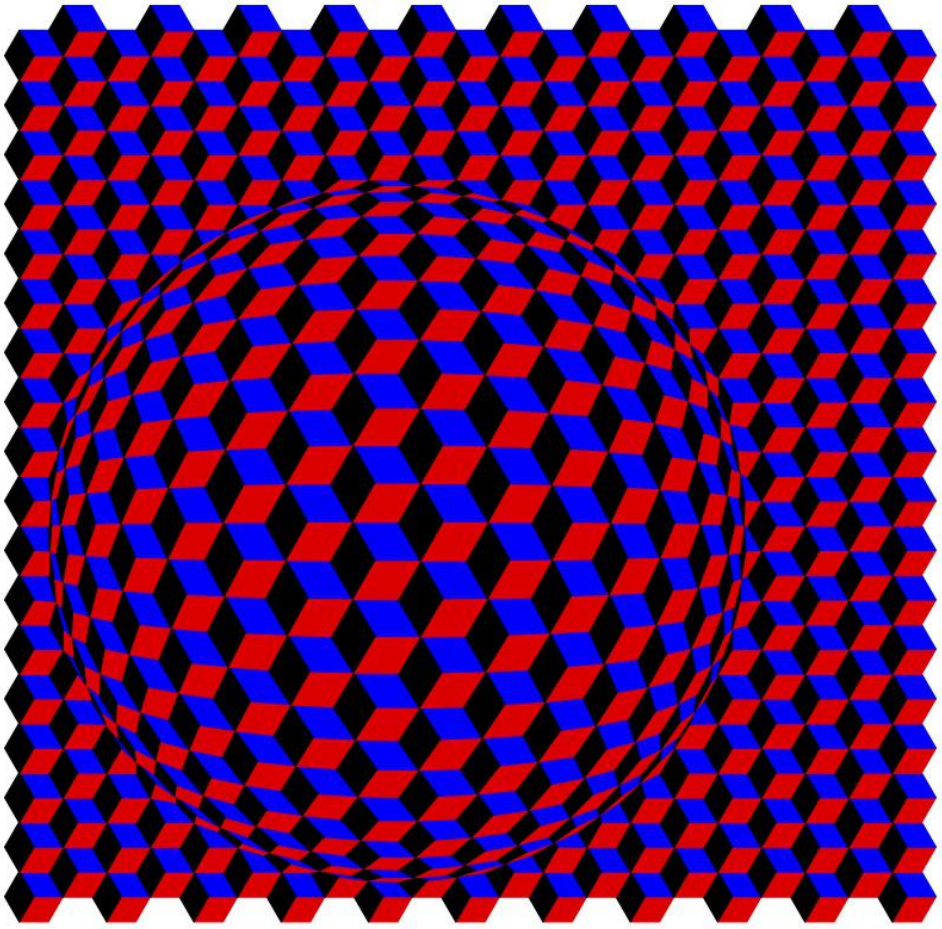
Le tableau au début de l'énoncé a été généré avec les paramètres
— `inf_gauche = -305`
— `sup_droit = 305`
— `longueur = 20`
— `col1 = 'blue'`
— `col2 = 'black'`
— `col3 = 'red'`
— `centre = (-50, -50, -50)`
— `r = 200`

Le second exemple en début d'énoncé est produit avec les mêmes paramètres, mis à part :
— `(col1, col2, col3) = ('white', 'black', 'grey')`
— `centre = (-100, -100, 0)` et

— `rayon = 300`

L'exemple ci-dessous a les mêmes paramètres que celui donné au début, mis à part :

— `centre = (-50, -50, 0)` et
— `rayon = 240`



pavage avec `centre = (-50, -50, 0)` et `rayon = 240`

8.1.3 Remise et évaluation du projet

MAIS COMMENT ÉVALUER UN PROJET ?

De façon binaire nous pourrions dire qu'un programme informatique est bon s'il fait ce qu'on lui demande et sinon il est incorrect. Ce serait un peu court surtout après avoir parlé de toutes les règles de bonnes pratiques que nous avons vues.

En fait, un projet tel que le projet Vasarely est évalué selon plusieurs critères de correction (programme correct, qui fait bien son travail) et de style (code qui suit les règles de bonnes pratiques).

Donnons ici des recommandations pour évaluer un projet de programmation niveau débutant.

Principes de base

La réalisation d'un projet informatique comprend plusieurs aspects qui font l'objet de choix qui doivent généralement être justifiés, ou de mise en application, généralement sous forme de programme structuré.

Tout d'abord, il est important que l'évaluation soit le résultat d'une notation

— positive : qui consiste à donner des points quand certains aspects sont présents dans la solution, et
— négative : qui consiste à enlever des points sur certains aspects qui sont absents ou incorrects.

Il convient aussi in fine d'avoir une évaluation globale où un projet qui *fonctionne* se voit *naturellement* attribuer des points même si plusieurs problèmes existent dans la réalisation.

Le mieux pour obtenir une telle note qui est le fruit d'une évaluation **négative et positive**, est de découper la note finale en sous notes, qui chacune évalue un aspect, qui de même fait l'objet d'une évaluation *négative et positive* avec évaluation globale.

Proposition

Ci-dessous la découpe de la note à donner au projet, que nous vous demandons d'appliquer dans vos évaluations pour obtenir une notation uniforme pour tous. Le nombre de points pour chaque catégorie est proposée pour une note finale sur 24 points. Comme il s'agit d'un projet pour débutant, les structures de données et les méthodes pour résoudre les problèmes sont données et ne nécessitent pas de recherche pour l'apprenant ; aucune note n'est donc donnée pour ces deux aspects.

Grille de notation du projet (sur 24 points en référence au 24 points obtenu dans le MOOC pour cette partie)

- **Découpe (4 points)** [La découpe du code global avec commentaires] initiaux, et respect de l'ordre des parties : import des modules externes, suivi des définitions des constantes globales, suivi des définitions des fonctions et enfin, code global.
  - 4 points si les quatre parties sont présentes et l'ordre est respecté ;
  - 3 points si une des quatre parties manque ou est dans le mauvais ordre ;
  - 2 points si deux des quatre parties manquent ou sont dans le mauvais ordre ;
  - 1 points si trois des quatre parties manquent ou sont dans le mauvais ordre ;
  - 0 point si rien n'est satisfaisant.
- **Structuration en fonctions (3 points)** [La structuration en] fonctions de bonne taille (25 lignes maximum) et cohérentes (pas de fonctions définies mais non appelées par exemple) est réalisée :
  - 3 points si la structuration est satisfaisante ;
  - 2 points si des fonctions sont définies, mais certaines sont incohérentes soit sont trop longues ;
  - 1 points si des fonctions sont définies, mais certaines sont incohérentes et d'autres sont trop longues ;
  - 0 point si aucune fonction n'est définie ou une certaine structuration en fonction existe mais ne respecte pas du tout les règles de bonnes pratiques.
- **Bonnes pratiques (4 points)** [Les bonnes pratiques sont] respectées en matière d'utilisation des noms des constantes, variables, fonctions. . . , de l'indentation. . . :
  - 4 points si toutes les bonnes pratiques sont respectées ;
  - 3 points si les noms des constantes, variables et fonctions respectent les bonnes pratiques, mais l'indentation (4 caractères par incrément) n'est pas respectée ;
  - 2 points si les règles sont globalement repectées à quelques exceptions près à la fois au niveau indentations et noms de constantes, variables ou fonctions ;
  - 0 point si les règles de bonnes pratiques en matière d'encodage ne sont globalement pas respectées.
- **Commentaires (4 points)** [Commentaires et docstrings sont bien] présents : le docstring initial possède les informations suivantes : identité de l'auteur, date, ce que fait le programme, les fonctions possèdent un docstring décrivant les paramètres, ce que fait la fonction aux paramètres (par défaut, ils ne sont pas modifiés) et le résultat de la fonction, des commentaires pertinents existent quand c'est utile à la compréhension
  - 4 points si commentaires et docstrings sont présent et respectent les règles de bonnes pratiques ;
  - 3 points si un des éléments du docstring initial ou des commentaires est manquant ou non satisfaisant ;
  - 2 points si le docstring initial et les commentaires sont manquants ou non satisfaisants, mais que les fonctions ont des docstrings corrects ;
  - 0 point si les trois éléments docstring initial, commentaires et docstrings de fonctions sont manquants ou non satisfaisants.
- **Consignes (2 points)** : (voir énoncé du projet)
  - 2 points si les consignes du projet sont respectées ;
  - 1 point si la moitié des consignes est respectée ;
  - 0 point si auncune consigne n'est respectée.
- **Résultat (7 points)** : Le programme fonctionne correctement.
  - 7 points si c'est le cas ;
  - 6 points si le programme donne les bons résultats sur la plupart des jeux de données ;
  - 4 points si le programme donne le bon résultat sur au moins un jeu de données ;
  - 2 points si le programme s'exécute sans échec sur au moins un jeu de données ;
  - 0 point si le programme ne s'exécute pas sans échec.

Explication sur l'évaluation

En plus des points attribués, il est essentiel pour l'évaluateur d'expliquer pourquoi telle ou telle note a été donnée, pour que l'auteur du projet puisse comprendre les notes données et puisse s'améliorer les fois suivantes.

ET COMMENT ETAIT ÉVALUÉ UN PROJET ?

Dans le cadre des trois premières sessions du cours « Apprendre à Coder avec Python », l'évaluation de chaque projet, se faisait en deux temps :

— d'une part par une évaluation par les pairs
— d'autre part, par une auto-évaluation du projet après éventuelles améliorations.

En pratique, la remise du projet et son évaluation étaient faites en **6 étapes** :

1. Après l'avoir réalisé, une soumission du projet éventuellement avec un fichier supplémentaire montrant un résultat.
2. Chaque apprenant devait évaluer trois projets d'autres étudiants choisis au hasard.
3. Chacun recevrait les notes et commentaires sur son propre projet venant d'au moins deux autres de vos pairs (deux autres apprenants).
4. À la lumière des commentaires donnés par les pairs, chacun corrigeait et soumettait une version améliorée de son projet.
5. et réalisait une auto-évaluation, c'est-à-dire chacun évaluait son propre projet.

Le document consignes pour la notation, téléchargeable ici reprend les explications et détaille chaque notation et demande de commenter chaque note donnée.

La note finale du travail sera la somme des évaluations par les pairs (24 points) et de votre auto-évaluation (24 points).

Temps et délais pour réaliser le projet

L'estimation du temps moyen que va vous mettre le projet Vasarely (en supposant que vous utilisez la fonction `deformation` qui vous est fournie), est très variable de quelque heures à plusieurs dizaines d'heures. réalisation.

DÉTAILS PRATIQUES SUR L'ÉVALUATION

Les paragraphes qui suivent vous permettent de réaliser :

— d'une part l'évaluation des projets de trois de vos pairs
— et d'autre part, quand vous aurez reçu deux évaluations de votre propre projet (d'autres pairs), une auto-évaluation de votre projet.

DERNIERS CONSEILS AVANT DE FAIRE LES ÉVALUATIONS

L'évaluation par les pairs avec les commentaires qui l'accompagnent ainsi que l'auto-évaluation peuvent être très utiles pour l'apprentissage de la programmation,.

Pour que cela soit le cas, lors des différentes phases de notation, essayez d'être :

— objectif dans vos évaluations,
— clairs et précis dans vos commentaires,
— mais toujours courtois et bienveillants

comme vous voudriez que les autres le soient pour vous-même.

Bon travail !

8.1.4 Galerie des oeuvres produites par le projet

RAPPEL DU CONTEXTE

Lors des 3 premières sessions du MOOC « Apprendre à coder avec Python » le projet consistait à réaliser
- une oeuvre d'art géométrique (Op Art)
- sous la forme d'un pavage hexagonal déformé par une sphère.

Les liens ci-dessous est un recueil des résultats lors des 3 sessions où ce projet a été proposé, finis ou en cours, parfois non conformes à l'énoncé mais pouvant donner un effet intéressant, ou parfois beaucoup plus sophistiqués que ce qui était demandé, au gré de l'imagination et de l'expertise de chacun.

Grand merci à toutes et tous pour le plaisir que vous nous avez donné à la vision de ces réalisations !

Isabelle, Sébastien, Thierry et L'équipe Pédagogique

GALERIE DES OEUVRES

- Les oeuvres de la session 3
- Les oeuvres de la session 2
- Les oeuvres de la session 1

CHAPITRE 9

Annexes

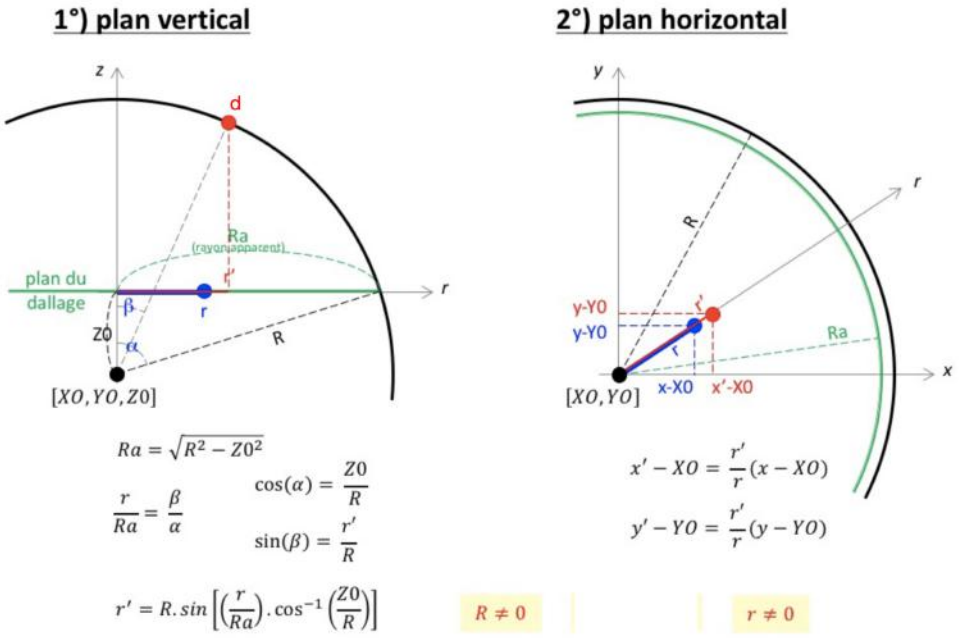
— Explications sur la déformation du projet Vasarely
— Aide-mémoire
— Petit manuel des bonnes pratiques Python

Projet Vasarely - Explications de la déformation

Crédit

La méthode et les schémas explicatifs sont de Thierry Duquenoÿ.

Explications



Pour les calculs, les points du pavage sont donnés avec des coordonnées en trois dimensions (x, y, z).
Le pavage sans déformation se trouve sur le plan avec $z = 0$ (plan x, y). Une sphère est définie par son centre et son rayon. Cette sphère va définir la déformation.
L'idée de base est que la surface déformée va se coller sur la sphère. Tous les points jusqu'aux bords de la déformation ne subissent aucune déformation. Le point au centre de la déformation va changer de hauteur (coordonnée z), mais ne changera pas non plus de coordonnées x et y. Par contre tous les autres points, à l'intérieur de la sphère vont avoir leurs trois coordonnées modifiées par la déformation. Le schéma ci-dessous explique comment les nouvelles coordonnées d'un point déformé seront calculées.
Nous supposons que le point avant déformation est à l'intérieur de la sphère de déformation et que le centre de déformation est sous le plan $z = 0$. Si le centre est au dessus, il suffit d'inverser le signe de z' après les calculs.
Regardons d'abord le plan vertical.

Sur le schéma, le centre de la déformation est en coordonnées $(X0, Y0, Z0)$ et le rayon vaut R.
Le rayon apparant Ra correspond au rayon de la calote de sphère apparent : d'après Pythagore, $Ra = \sqrt{R^2 - Z0^2}$.
Le point avant déformation (x, y, z) est tel que le rapport entre sa distance r entre au point $x = y = z = 0$ et la valeur Ra doit être égale après déformation au rapport entre l'angle β formé par l'axe $x = y = 0$ et le point d après déformation et l'angle α formé par l'axe $x = y = 0$ et le segment passant par le centre de la sphère $(X0, Y0, Z0)$ et par le point d'intersection entre le plan $z = 0$ et la sphère.
Ayant r', la distance entre la projection du point $d = (x', y', z')$ sur le plan $z = 0$;
Nous avons :

$$\frac{r}{Ra} = \frac{\beta}{\alpha}$$
$$cos(\alpha) = \frac{Z0}{R}$$
$$sin(\beta) = \frac{r'}{R}$$
$$r' = R.sin((\frac{r}{Ra}).cos^{-1}(\frac{Z0}{R}))$$

On peut donc voir avec le plan horizontal que :

$$x' = X0 + \frac{r'}{r}(x - X0)$$
$$y' = Y0 + \frac{r'}{r}(y - Y0)$$

Et à nouveau avec le plan vertical :

$$\beta = sin^{-1}(\frac{r'}{R})$$
$$z' = Z0 + R.cos(\beta)$$

Aide-mémoire PYTHON3 - v3.9.1

Fonctions prédéfinies

- `abs(x)` : valeur absolue de x
- `int(x)` : valeur x convertie en entier
- `float(x)` : valeur x convertie en réel
- `str(x)` : valeur x (int ou float), convertie en str
- `list(x)` : valeur x convertie en liste
- `tuple(x)` : valeur x convertie en tuple
- `dict(x)` : séquence de couples x convertie en dictionnaire
- `set(x)` : x converti en ensemble
- `help(x)` : aide sur x
- `dir(x)` : liste des attributs de x
- `type(x)` : type de x
- `print(...)` : imprime
- `input(x)` : imprime le string x et lit le string qui est introduit
- `round(x [,ndigits])` : valeur arrondie du float x à ndigits chiffres (par défaut 0)
- `range([start], stop, [step])` : retourne une suite d'entiers
- `sorted(s)` : retourne une liste avec les éléments de s triés

Gather, scatter et keyword arguments

- `def fun(*args)` : *gather* des arguments en un tuple args
- `fun(*s)` : *scatter de la séquence s lors de l'appel

Opérations et méthodes sur les séquences (str, list, tuples)

- `len(s)` : longueur de la séquence s
- `min(s), max(s)` : élément minimum, maximum
- `sum(s)` : (ne fonctionne pas pour les string) : somme de tous les éléments de la séquence s (valeur numérique)
- `s.index(value, [start, [stop]])` : premier indice de value dans s[start :stop]
- `s.count(sub [,start [,end]])` : nombre d'occurrences sans chevauchement de sub dans `s[start:end]`
- `enumerate(s)` : construit une séquence de couples dont le ième élément (à partir de 0) vaut le couple (i, s[i])
- `zip (a,b), zip(a,b,c)`, ... : construit une séquence de couples, resp. triples, ..., dont le ième élément reprend le ième élément de chaque séquence a, b [,c]

Méthodes sur les chaînes de caractères (str)

- `s.lower(), s.upper()` : string avec caractères en minuscules respectivement en majuscules
- `s.islower()`, `s.isdigit()`, `s.isalnum()`, `s.isalpha()`, `s.isupper()` : vrai si s n'est pas vide et n'a (respectivement) que des minuscules, des chiffres, des car. alphanumériques, alphabétiques, majuscules
- `s.find(sub [,start [,end]])` : premier indice de s où le sous string sub est trouvé dans `s[start:end]`
- `s.replace( old, new[, co])` : copie de s en remplaçant toutes les (ou les co premières) occurrences de old par new.
- `s.format(...)` : copie de s après formatage
- `s.capitalize()` : copie de s avec première lettre en majuscule
- `s.strip()` : copie de s en retirant les *blancs* en début et fin
- `s.join(t)` : crée un str qui est le résultat de la concaténation des éléments de la séquence de str t chacun séparé par le str s
- `s.split([sep [,maxsplit])` : renvoie une liste d'éléments séparés dans s par le caractère sep (par défaut *blanc*) ; au maximum maxsplit séparations sont faites (par défaut l'infini)

Opérateurs et méthodes sur les listes

- `s.append(v)` : ajoute un élément valant v à la fin de la liste
- `s.extend(s2)` : rajoute à s tous les éléments de la liste s2
- `s.insert(i,v)` : insère l'objet v à l'indice i
- `s.pop([i])` : supprime l'élément d'indice i de la liste (par défaut le dernier) et retourne la valeur de l'élément supprimé
- `s.remove(v)` : supprime la première valeur v dans s
- `s.reverse()` : renverse l'ordre des éléments de la liste, le premier et le dernier élément échangent leurs places, ...
- `s.sort(key=None, reverse=False)` : trie s en place
- `s.copy()` : *shallow* copie superficielle de s
- `del s[i], del s[i:j]` : supprime un ou des éléments de s

Méthodes sur les dictionnaires (dict)

- `d.clear()` : supprime tous les éléments de d
- `d.copy()` : *shallow* copie de d
- `{}.fromkeys(s,v)` : crée un dict avec les clés de s et la valeur v
- `d.get(k [,v])` : renvoie la valeur d[k] si elle existe v sinon
- `d.items()` : liste des items (k,v) de d
- `d.keys()` : liste des clés
- `d.pop(k [,v])` : enlève d[k] s'il existe et renvoie sa valeur ou v sinon
- `d.popitem()` : supprime un item arbitraire (k,v) et retourne l'item sous forme de tuple
- `d.setdefault(k [,v])` : d[k] si elle existe sinon v et rajoute d[k]=v
- `d.update(s)` : s est une liste de tuples que l'on rajoute à d
- `d.values()` : liste des valeurs de d
- `del d[k]` : supprime l'élément de clé k de d

Méthodes sur les ensembles (set)

- `s = set(v)` : initialise s : un set contenant les valeurs de v
- `s.add(v)` : ajoute l'élément v au set s (ne fait rien s'il y est déjà)
- `s.clear()` et `s.copy()` : idem dictionnaires
- `s.remove(v)` : supprime l'élément v du set (erreur si v n'est pas présent dans s)
- `s.discard(v)` : si v existe dans s, le supprime
- `s.pop()` : supprime et renvoie un élément arbitraire de s

Modules

- `math` : accès aux constantes et fonctions mathématiques (`pi`, `sin()`, `sqrt(x)`, `exp(x)`,`floor(x)` (valeur plancher), `ceil(x)` (valeur plafond), ...) : exemple : `math.ceil(x)`
- `copy: copy(s), deepcopy(s)` : *shallow* et *deepcopy* de s

Méthodes sur les fichiers

- `f = open('fichier')` : ouvre 'fichier' en lecture (autre paramètres possibles : 'w'(en écriture), 'a'(en écriture avec ajout), encoding ='utf-8' : encodage UTF-8)
- `with open('fichier'...) as f` : ouvre 'fichier' pour traitement à l'intérieur du with
- `f.read()` : retourne le contenu du fichier texte f
- `f.readline()` : lit une ligne
- `f.readlines()` : renvoie la liste des lignes de f
- `f.write(s)` : écrit la chaîne de caractères s dans le fichier f
- `f.close()` : ferme f

CC BY SA

2.09.2023

Quelques bonnes pratiques de programmation (Python)

Traduction d'un problème en programme

1. *Analysez le problème*

- Identifiez clairement ce que sont les **données fournies**, ainsi que les **résultats et types** attendus à l'issue du traitement.
- Formalisez une **démarche générale de résolution** par une séquence d'opérations simples.
- Vérifiez que vous **envisagez tous les cas** de figure (en particuliers les cas "limites").

2. *Découpez votre problème en fonctions*

- Chaque fonction doit réaliser ***une* tâche** clairement identifiée.
- Limitez les fonctions à **25 lignes** maximum, sauf dans des cas exceptionnels.
- **Evitez la redondance** dans le code (copier/coller). Si cela arrive, c'est qu'il manque soit une fonction, soit une boucle, soit que des tests conditionnels peuvent être regroupés.
- N'utilisez **pas de variables globales**.
- Veillez à ce que tous les paramètres et variables d'une fonction soient **utilisés** dans cette fonction.
- Pour une fonction qui renvoie un résultat, si cela ne complique pas trop le code, organisez le pour qu'il ne contienne qu'**un seul** `return`, placé comme dernière instruction de la fonction. Pour une fonction qui ne renvoie pas de résultat, vous ne devez pas mettre de `return` (il y en aura un implicitement à la fin de l'exécution de la fonction).
- Ne modifiez pas les *paramètres*.□ Exemple: Si vous recevez une borne inférieure `first` et une supérieure `last` et que vous devez itérer de la première à la dernière, n'incrémentez pas `first` dans la boucle, car la signification n'en serait plus claire ; créez plutôt une variable locale `pos` initialisée à `first`.
- Sauf si la fonction a comme but de modifier la (structure de) données reçue en paramètre ; dans ce cas la fonction ne renvoie pas de valeur.

3. *Testez le code au fur et à mesure du développement*

- Créez des **scénarios de test**, pour lesquels vous choisissez les données fournies et vous vérifiez que le résultat de la fonction est conforme à ce que vous attendez.
- Vérifiez les cas particuliers et les *conditions aux limites*. □ Exemples: Pour le calcul d'une racine carrée, que se passe-t-il lorsque le paramètre est un nombre négatif ?

Programmation

1. *Style de programmation*

- N'utilisez pas les instructions `break` ou `continue`
- Utilisez la forme raccourcie `if(is_leap_year(2022))` plutôt que la forme équivalente `if(is_leap_year(2022)==true)`
- Utilisez la forme `return <expression booléenne>` plutôt que la forme équivalente

```
if <expression booléenne>:
  res = True
else:
  res = False
return res
```

- N'exécutez pas plusieurs fois une fonction alors qu'une exécution suffit en retenant le résultat.
- Précisez le domaine de validité des paramètres.

Programmation (suite)

2. *Quelques erreurs classiques*

- Vous essayez d'utiliser une variable avant de l'avoir **initialisée**.
- L'**alignement des blocs** de code n'est pas respecté.
- Vous oubliez de fermer un fichier que vous avez ouvert.

Nommage de variables, fonctions, *etc.*

1. *Utilisez une convention de nommage*

`joined_lower`	pour les **variables** (attributs), et **fonctions** (méthodes)
`ALL_CAPS`	pour les **constantes**

2. *Choisissez bien les noms*

- Donnez des noms de variables qui expriment leur contenu, des noms de fonctions qui expriment ce qu'elles font (cf. règles de nommage ci-dessus).
- Évitez les noms trop proches les uns des autres.
- Utilisez aussi systématiquement que possible les mêmes genres de noms de variables.
  □ Exemples: `i`, `j`, `k` pour des indices, `x`, `y`, `z` pour les coordonnées, `max_length` pour une variable, `is_even()` pour une fonction, etc.

Style et documentation du code

1. *Soignez la clarté de votre code*

... c'est la première source de documentation.

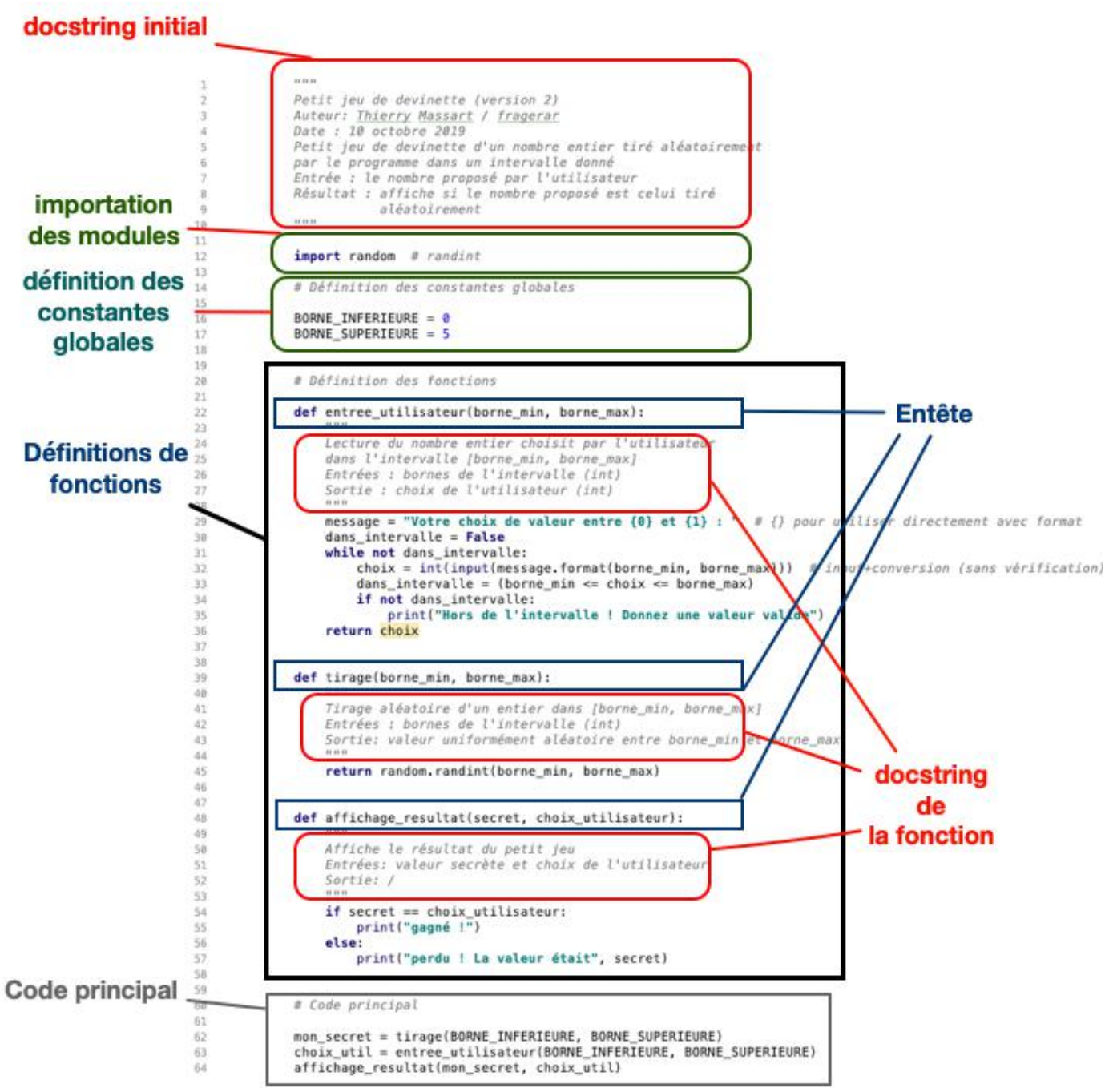
- Utilisez les `docstrings` dans chaque fonction pour :
  - brièvement décrire ce que fait la fonction, **pas comment elle le fait**, et préciser ses entrées et sorties.
  - décrire les arguments des fonctions.
- Soignez les indentations (2 à 4 espaces chacune) et la gestion des espaces et des lignes blanches (deux lignes blanches avant et entre chacune des définitions de fonction globales ; une ligne blanche pour mettre en évidence une nouvelle partie dans une fonction),
- Il faut commenter le code à bon escient et avec parcimonie. Évitez d'indiquer le *fonctionnement* du code dans les commentaires. □ Exemples: Avant l'instruction "`for car in line:`", ne pas indiquer qu'on va boucler sur tous les caractères de la `line`...
- Évitez de paraphraser le code. N'utilisez les commentaires que lorsque la fonction d'un bout de code est difficile à comprendre.

Structure d'un programme Python

1. *Voir verso*

Quelques bonnes pratiques de programmation (Python)

Structure d'un programme Python


docstring initial
importation des modules
définition des constantes globales
Définitions de fonctions
Entête
docstring de la fonction
Code principal
"""
Petit jeu de devinette (version 2)
Auteur: Thierry Massart / fragerar
Date : 10 octobre 2019
Petit jeu de devinette d'un nombre entier tiré aléatoirement
par le programme dans un intervalle donné
Entrée : le nombre proposé par l'utilisateur
Résultat : affiche si le nombre proposé est celui tiré
aléatoirement
"""
import random # randint
# Définition des constantes globales
BORNE_INFERIEURE = 0
BORNE_SUPERIEURE = 5
# Définition des fonctions
def entree_utilisateur(borne_min, borne_max):
"""
Lecture du nombre entier choisit par l'utilisateur
dans l'intervalle [borne_min, borne_max]
Entrées : bornes de l'intervalle (int)
Sortie : choix de l'utilisateur (int)
"""
message = "Votre choix de valeur entre {0} et {1} : " # {} pour utiliser directement avec format
dans_intervalle = False
while not dans_intervalle:
choix = int(input(message.format(borne_min, borne_max))) # input+conversion (sans vérification)
dans_intervalle = (borne_min <= choix <= borne_max)
if not dans_intervalle:
print("Hors de l'intervalle ! Donnez une valeur valide")
return choix
def tirage(borne_min, borne_max):
"""
Tirage aléatoire d'un entier dans [borne_min, borne_max]
Entrées : bornes de l'intervalle (int)
Sortie: valeur uniformément aléatoire entre borne_min et borne_max
"""
return random.randint(borne_min, borne_max)
def affichage_resultat(secret, choix_utilisateur):
"""
Affiche le résultat du petit jeu
Entrées: valeur secrète et choix de l'utilisateur
Sortie: /
"""
if secret == choix_utilisateur:
print("gagné !")
else:
print("perdu ! La valeur était", secret)
# Code principal
mon_secret = tirage(BORNE_INFERIEURE, BORNE_SUPERIEURE)
choix_util = entree_utilisateur(BORNE_INFERIEURE, BORNE_SUPERIEURE)
affichage_resultat(mon_secret, choix_util)

Index

+, 40
=, 42
#croisillon, 47
%, 37
évaluation paresseuse, 80
évaluations de votre apprentissage, 9
évaluer le projet, 314, 326

add, 253
affectation, 42
aide-mémoire, 57, 331
and, 76
append, 196
as, 233
assert, 139
assignation, 42
assignation multiple, 65
associativité, 38, 76

Bonnes pratiques Python, 157, 331

capitalize, 212
caractère d'échappement "\", 64
caractère de continuation "\", 155
carré magique, 242
Catalan, 171
chaîne de caractères, 39, 181, 183
chaîne de caractères multiligne, 64
chr, 186
clé, 255
classe, 200
clear, 253, 266
close, 232, 234
code "propre", 151
codification utf-8, 185
commentaire, 46, 47
compréhension, 225
concaténation, 40
Conjecture de Syracuse, 97
constante, 44
contrôle de type, 138
copie
    superficielle, profonde, 225
copy, 266
copy(), 214
csv, 293

déboguer, 49
déformation du projet Vasarely, 331
deepcopy, 266
del, 214
diagramme d'état, 45
diagramme d'état, 139, 209
dict, 266
dictionnaire, 255
dictionnaire persistant, 293
discard, 253
division en nombre flottant, 37
division entière, 37
docstring, 46
documentation Python, 31

elif, 71
else, 71
ensemble, 253
entrée, 44
enumerate, 211
escape game, 301
espaces de discussion, 10
eval, 201
exponentiation, 37
expression, 36
extend, 196

f-strings, 154
False, 71
Fibonacci, 142
Fibonaccisuite de, 104
fichier, 234
    méthode ; open, read, readline, write, close, 234
fichier csv, 291
find, 212
float, 36, 63
fonction

corps, entête, paramètre, argument, 133
définition, appel, return, 131
fonction booléenne, 135
fonction lambda, 230
fonction prédéfinie, 130
for, 101, 184, 233, 253, 255
format, 154
forums, 8, 10
fromkeys, 266

gather, 192
get, 266

help, 56

identificateur, 78
if, 71
in, 184, 233, 253, 255, 266
incrémentation, 65
indentation, 79
index, 211
informatique, 31
input, 44
insert, 214
instruction conditionnelle, 71
instruction répétitive, 95, 101
int, 36, 63
isalnum, 212
isalpha, 212
isdigit, 212
isinstance, 57
islower, 212
items, 266

jeu d'évasion, 301
join, 227
json, 293

keys, 266
keyword, 78

len, 40
ligne de continuation, 155
Linux, 22
liste, 182, 183
liste persistante, 293
lois de De Morgan, 77
lower, 212

méthode, 200, 209
MacOs, 21
math, 53
matrice, 241
initialisation, 241
min, 211
mode interactif, 37
modulo, 37

mot-clé, 78

Nicolas Boileau, 297
None, 135
not, 76

objet, 200
opérateur arithmétique
+, -, *, //, /, **, %, 37
opérateur de mise à jour, 65
opérateur ensembliste, 253
opérateur logique, 76
opérateur relationnel
==, <=, >=, !=, <, >, 73
open, 233, 234
or, 76
ord, 186

paramètre
passage, effectif, formel, argument, 133
valeur par défaut, 162
pass, 126
pavé hexagonal, 59
pep 20, 158
pep 8, 152
polymorphisme, 138
pop, 214, 253, 266
portée, 141
print, 44
priorité, 38, 76
projet Chateau, 301
puissance 4, 243
Python Tutor, 30, 45

racines d'une équation du second degré, 75
random, 81
rassembler, 192
read, 234
readline, 232, 234
remove, 214, 253
replace, 212
reverse, 214

sémantique, 44
séparer, 193
séquence
initialisation, 241
initialisation, lecture, 201
séquence de fonctions, 273
scatter, 193
set, 253
setdefault, 266
slice, 195
slicing, 195
sort(), 214
sorted, 214
sortie, 44

split, 216
strip, 212, 232
sudoku, 282, 286
sum, 211
syntaxe, 44
syntaxe de l'instruction for, 102
syntaxe de l'instruction if, 78
syntaxe de l'instruction while, 96

table de vérité des opérateurs logiques, 76
tableur, 291
tester, 37
Thonny, 16
Traceback, 141
tranche, 195
Trinket, 23
True, 71
tuple, 182, 183
turtle, 53, 57
type, 37, 57
type entier, 37
type flottant, 37
type fractionnaire, 37

Ubuntu, 22
update, 266
upper, 212
UpyLaB, 4, 27, 29, 49

valeur, 36, 255
values, 266
variable, 41
    locale, globale, 139

while, 95
Windows, 20
with, 233
write, 234

yin-yang, 136, 161

zip, 211

À propos du cours

Vous avez un ordinateur, désirez apprendre à coder et êtes totalement ou partiellement débutant dans le domaine; vous êtes étudiant, professeur ou simplement une personne qui sente l'envie ou le besoin d'apprendre la programmation de base; ce cours utilise Python 3 comme clé pour vous ouvrir la porte de cette connaissance informatique.

Ce cours est orienté vers la pratique, et propose un matériel abondant pour couvrir l'apprentissage de la programmation de base, d'une part en montrant et expliquant les concepts grâce à de nombreuses capsules vidéo courtes et des explications simples, et d'autre part en vous demandant de mettre ces concepts en pratique d'abord de façon guidée et ensuite autonome. Plusieurs quiz, un projet individuel, et de nombreux exercices à réaliser et validés automatiquement avec notre outil UpyLaB intégré au cours, vous permettent de polir et ensuite de valider votre apprentissage.

Format du cours

Le cours s'étale sur 9 semaines et propose 3 parcours d'apprentissage et correspond à un travail hebdomadaire de 6 à 12 heures avec un projet évalué par les pairs. Si vous ne pouvez y consacrer 6h par semaine un parcours à allure libre est possible, mais sans projet ni attestation finale délivrée par FUN.

Les enseignants

SÉBASTIEN HOARAU

Sébastien Hoarau est maître de conférences à l'Université de la Réunion (UR) où, depuis plus de 20 ans, il enseigne la programmation aux étudiants de première année scientifique. Sa pédagogie est axée sur la pratique : quiz en cours, projets, utilisation de plateformes ludiques.

THIERRY MASSART

Thierry Massart est professeur à l'Université Libre de Bruxelles (ULB) où, depuis plus de 25 ans, il enseigne la programmation principalement aux étudiants de Sciences Informatique et de l'école Polytechnique de l'ULB. Il leur propose une pédagogie active orientée vers la pratique.

JEAN OLGIATI

Formé en ingénierie, graphisme et pédagogie, Jean travaille à l'Université libre de Bruxelles et à la Scientothèque, association qui promeut l'égalité des chances par les sciences. Il participe à des projets de soutien aux enseignants qui investissent le champ du numérique, et anime auprès de jeunes un atelier sur la programmation et la robotique.. Il a découvert Python grâce au présent MOOC et a contribué à définir le projet de fin de cours.

ISABELLE POIRIER

Isabelle est professeur agrégé de mathématiques et enseigne celles-ci depuis plus de 15 ans en établissement secondaire dans le sud de la France. Autodidacte en programmation (en particulier grâce à sa participation à plusieurs MOOC), elle met à profit son sens de la pédagogie et sa propre expérience pour venir en aide aux apprenants sur le forum et améliorer l'approche didactique proposée au bénéfice d'un meilleur apprentissage pour tous.